Parallel Finite Element Method with FEAP 8.2

Bachelor thesis by Björn Müller
August 2008
Supervision: Dipl.-Ing. Steffen Bauer

TECHNISCHE UNIVERSITÄT DARMSTADT

fnb Fachgebiet Numerische Berechnungsverfahren im Maschinenbau

Abstract

The present Bachelor Thesis deals with the parallelised version of the Finite Element (FE) software tool FEAP. The first part represents an introduction to the theoretical background of the underlying continuum mechanics equations, the Finite Element Method, the solving of the emerging linear systems of equations and the basics of parallelisation. The following chapters describe several aspects of the software packages itself. This includes the required software installation steps in two different environments (Linux and the UNIX derivate IBM AIX) and the validation of the obtained results. In contrast, the second part focuses on the productive application of FEAP. The corresponding topics cover the development of software tools for a convenient handling in common situations and the examination of the actual speed-up for a practical example with different solving algorithms. Finally, the work is concluded by an evaluation of the obtained results and a brief discussion of suitable fields of application.

Statutory declaration

I hereby declare that I wrote this thesis independently and without use of other sources than acknowledged. Passages taken literally or analogously from published or non published sources are marked as such. Drawings or figures in this thesis have either been created by myself or their source is given. This work has not been presented to an examination board in this or a related form previously.

Darmstadt, September 30, 2008

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

Numerical simulations have become an important tool in many branches of science in the recent past. Especially for structural mechanics applications, corresponding techniques nowadays represent a third standard approach in addition to the classical experimental and analytical concepts. In this context, the so called Finite Element Method (FEM) has become a quasi-standard and a wide variety of application programs exist. One of these programs, FEAP[1], is subject to this thesis. More precisely, its recent ability to make use of multiple processors for the processing of computations and the emerging implications will be addressed in detail.

1.1 Motivation

Since the beginning of the modern computer industry, the available computing performance for given hardware costs has been rising exponentially. For the illustration of this fact *Moore's law* (dating back to 1965) has often been cited which originally states that

> "The complexity for minimum component costs has increased at a rate of roughly a factor of two per year [...]. Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years." [9]

This means that the density at minimum cost per basic component (most notably transistors) on an integrated component doubles approximately every two years. As a matter of fact, the hardware performance has been growing rapidly for the last 40 years which made the simulation of more and more complex problems possible. This development has been supported by the evolution of more powerful numerical algorithms. But on the other hand, this trend cannot be maintained forever due to physical limitations and the by now high stage of development of the available numerical methods. In fact, the implication that the performance of a single integrated circuit should also double every two years is not valid anymore which is underlined by the fact that the feasible clock rates have been hardly increasing in recent years (see [13]). The struggle for faster clock rates has instead been replaced by the use of multi-core processors (and thus multiple integrated circuits) which are available for the same cost. Consequently, the available computing performance for given hardware costs still doubles roughly every two years, but only if regarding the state-of-the-art multi-processor systems. This shows the need for numerical algorithms and tools which can make use of the full power of this parallel environment for the solving of more and more complex problems.

Standard programs generally run a single processor (*serial* execution) unless they are modified in order to run in so called *threads* on several processors at the same time. These modifications are often referred to as the *parallelisation of computation* and make extensive changes of the existing source code necessary. The efficiency of the resulting *parallelised* program highly depends on the amount of communication between separate threads as the number of processors increases. Nevertheless, this step is inevitable in almost any field dealing with numerical analysis for the purpose of keeping computing times as small as possible. Since version 8.2, the mentioned FEM tool FEAP supports parallelisation and thus follows the above-mentioned development.

1.2 Objective and outline

The subject of this thesis is quite versatile and covers extensive parts of the work with the parallelised version of FEAP 8.2 with the goal of offering a detailed overview of the major consequences of its adoption. This task ranges from installation instructions to the improvement of the manageability and the analysis of its performance. Of course, this also includes drawbacks, possible problems and the evaluation of the overall ease of use. In short, this means that this thesis should make a potential user able to efficiently use the software package while providing enough background information to permit a thorough understanding of the implications.

To this end, chapter 2 is dedicated to aspects forming the basis of computations performed with FEAP 8.2. This refers to the fundamentals of the FEM as well as the solving of the emerging problem and the basics of parallelisation. The software package itself is introduced in the subsequent chapter with focus on the general structure, installation steps and the usage in simple cases. The obtained parallel built of FEAP is then validated in chapter 4. During the completion of

[1] Finite Element Analysis Program – `http://www.ce.berkeley.edu/~rlt/feap/`

this task, several experiences have been made concerning the usability of the software package. The circumvention of the encountered drawbacks are thus subject to the following chapter. Chapter 6 concludes the preliminary considerations and proceeds to the usage of FEAP in a massive parallel environment. This especially includes performance analyses of the solving process and a discussion on the topic of load distribution. Finally, an overall evaluation of the acquired results will be given in the last chapter.

2 Fundamentals

In the course of this thesis, numerical analyses on continuum mechanics problems making use of a number of mathematical concepts, definitions and methods will be discussed. This includes the underlying continuum mechanical model, the corresponding *Finite Element* approach, the solving of the resulting linear system of equations and the background for the parallelisation of this process. The respective general ideas will be outlined briefly in this chapter while a more extensive introduction can be found in [11] which has also been the source for the following considerations if not otherwise stated.

2.1 Governing equations for continuum mechanics problems

Models for continuum mechanics problems generally base on the *momentum conservation law* which states that the temporal change of the momentum of a body has to be equal to the sum of all body and surfaces acting on the the body for all points in time. If the momentum of a body is defined as

$$p_i(t) = \int_V \rho(x,t) v_i(x,t) dV \tag{2.1}$$

(with the density $\rho(x,t)$ and the velocity $v(x,t)$), the differential formulation of this law yields

$$\frac{\partial(\rho v_i)}{\partial t} + \frac{\partial(\rho v_i v_j)}{\partial x_j} = \frac{\partial T_{ij}}{\partial x_j} + \rho f_i. \tag{2.2}$$

In this context, f_i denotes the acting volume forces and T_{ij} stands for the symmetric *Cauchy stress tensor* which describes the body's state of stress.

Another important characteristic of continuum mechanics problems is the strain of each point. If the acting forces only cause small deformations of the problem geometry, a linear strain-displacement relation can usually be applied. In this geometrically linear case the strain may be defined as

$$\epsilon_{ij} = \frac{1}{2}\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right) \tag{2.3}$$

and we will restrict ourselves to this formulation here. As a result, (2.2) can be written as

$$\rho \frac{D^2 u_i}{Dt^2} = \frac{\partial T_{ij}}{\partial x_j} + \rho f_i \tag{2.4}$$

while u_i denotes the displacement subject to x and t.

Equation (2.4) still contains the unknown stress tensor T_{ij}. Its components cannot be derived from basic conservative principles and thus some kind of material law has to be assumed. For linear elastic material behaviour *Hooke's law* is used most frequently because it can be applied for most materials if the encountered stresses are smaller than the corresponding yield stress. In compatible situations, the stress tensor is then defined by

$$T_{ij} = \lambda \epsilon_{kk} \delta_{ij} + 2\mu \epsilon_{ij} \tag{2.5}$$

with the material-dependent *Lamé constants* λ and μ. Both constants can be derived from the well-known *elasticity modulus* E and the *Poisson ratio* ν:

$$\lambda = \frac{E\nu}{(1+\nu)(1-2\nu)} \quad \text{and} \quad \mu = \frac{E}{2(1+\nu)}. \tag{2.6}$$

Stresses exceeding the elastic range of a material (*plastic deformations*) make several modifications of (2.5) necessary which will not be discussed here but can for example be found in [17].

The combination of (2.4) and (2.5) leads to the *Navier-Cauchy equations of linear elastic theory*:

$$\rho \frac{D^2 u_i}{Dt^2} = (\lambda + \mu) \frac{\partial^2 u_j}{\partial x_i \partial x_j} + \mu \frac{\partial^2 u_i}{\partial x_j \partial x_j} + \rho f_i \tag{2.7}$$

In association with appropriate boundary and initial conditions (e.g. prescribed displacements and/or stresses), (2.7) denotes a closed system of partial differential equations which can be used for the determination of the unknown displacements u_i.

If the problem itself is steady (e.g. because a static load is considered), (2.7) can be simplified to

$$0 = (\lambda + \mu)\frac{\partial^2 u_j}{\partial x_i \partial x_j} + \mu \frac{\partial^2 u_i}{\partial x_j \partial x_j} + \rho f_i \tag{2.8}$$

while the formulation of intitial conditions may now be omitted.

2.2 The Finite Element Method

Currently, no universal algebraic solutions to the partial differential equations introduced in 2.1 are known. Therefore, several concepts have been developed to approximate the solution of such problems. For structural mechanics problems, the Finite Element Method (FEM) is applied most frequently. We restrict ourselves to the steady case here and again refer to [11] for the modifications in the transient case. The FEM bases on the assumption that the solution of time-independent partial differential (e.g. (2.8)) can be approximated by

$$\Phi(x) \approx \varphi_0(x) + \sum_{k=1}^{N} c_k \varphi_k(x) \tag{2.9}$$

with arbitrary (but fix) functions φ_k vanishing on the Dirichlet boundaries of the problem domain Ω. The function φ_0 instead satisfies all Dirichlet boundary conditions (that is, prescribed displacements). The coefficients c_k are unkown and so far undefined. In the following, we will outline how the FEM defines these variables and how it is used in order to determine an adequate approximation of Φ.

2.2.1 Method of the weighted residuals

In the context of numerical solutions it can generally not be guaranteed that the underlying equations are fulfilled exactly which is why one introduces the *residual* R by writing the considered partial differential equation as

$$R = 0 \tag{2.10}$$

and inserting the ansatz (2.9). That way, R defines a measure for the current deviation from the exact solution. The claim that the integral mean value of R over the problem domain Ω vanishes leads to conditions to the unknown values $c_1, \ldots, c_N$. If this mean value is determined utilising N arbitrary (but linearly independent) *test functions* w_j which again vanish on all Dirichlet boundaries, this approach is called the *method of the weighted residuals* which can simply be written as

$$\int_\Omega R\omega_j d\Omega \stackrel{!}{=} 0 \quad \forall j = 1, \ldots, N. \tag{2.11}$$

2.2.2 Discretisation

Before (2.11) can be used to determine the unknown values, the problem domain has to be approximated in order to be able to handle it numerically with limited computational accuracy. To this end, the affected geometry is transformed into a gapless mesh of non-overlapping *finite elements* which might be triangles or quadrilateral structures in a two-dimensional case. Figure 2.1 shows an example of the corresponding process.

The description of the state of an element is usually achieved with the aid of piecewise polynomial *ansatz functions* which are formulated via designated local attributes $\Phi_1^i, \ldots, \Phi_p^i$ based on p node values of the i-th element. Along with functional *local shape functions* $N_1, \ldots, N_p$, this approximation can be written as

$$\Phi^i(x) = \sum_{j=1}^{p} \Phi_j^i N_j^i(x) \tag{2.12}$$

where the Φ_j^i denote N unknown values which have to be determined using the FEM. The topology of all elements is described by the *coincidence matrix*. It holds a list of all elements associated with its local node variables and hence allows

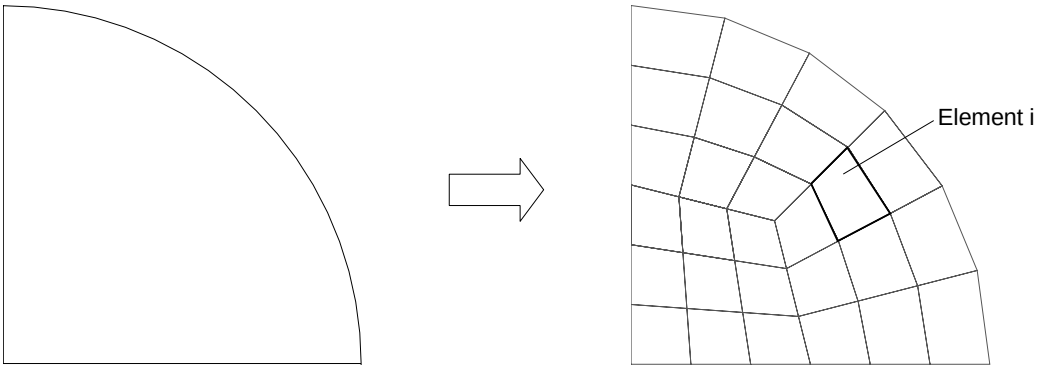


Figure 2.1: Example of the subdivision of a 2D problem domain into *finite elements*

the identification of all elements referring to a given variable. As a result, the *local view* specified by (2.12) in combination with the boundary conditions may be employed to recover a global representation of the approximated solution

$$\Phi(x) \approx \varphi_0(x) + \sum_{k=1}^{N} \Phi_k N_k(x), \tag{2.13}$$

where φ_0 is a function satisfying the Dirichlet boundary conditions (that is, prescribed displacements) and $N_k(x)$ is a suitable incorporation of the local shape functions $N_j^i(x)$ associated with the local counterparts of Φ_k. Now it becomes clear that (2.12) in fact leads to a specialised version of (2.9) where the Φ_k take the places of the c_k.

2.2.3 The Galerkin method and the resulting system of equations

One can think of numerous ways of chosing the test functions w_j in (2.11). If the ansatz functions φ_k are also applied as test functions, one speaks of the *Galerkin method* which forms the basis of the FEM. Having fixed the test functions in (2.11), solely the Φ_k remain unkown. In combination with an adequate numerical integration scheme and the coincidence matrix, all local formulations may be incorporated into a (preliminary) global linear system of equations for the unknown values.

The obtained system of equations cannot be solved in this original form as the boundary conditions have not been considered yet. Fortunately, this step is straightforward since prescribed displacements directly determine the values of the corresponding nodal attributes and prescribed stresses can be integrated into the right hand side of the resulting system

$$A\Phi = b. \tag{2.14}$$

Here, A denotes the symmetric, sparse, positive definite *stiffness matrix*, b the defines the *load vector* and Φ contains all local attributes $\Phi_1, \ldots, \Phi_N$.

2.3 Solution of linear systems of equations

In recent years, many powerful methods have been developed which are capable of solving bigger and bigger linear systems of equations which correspond to (2.14). Besides direct algorithms like the well-known *Gaussian elemination* which compute the exact solution (up to numerical precision), especially iterative methods are applied nowadays due to their superior performance on sparse matrices. In the following, four respective algorithms will be introduced according to [10].

2.3.1 Jacobi and Gauss-Seidel

The Jacobi and the Gauss-Seidel method both are classical iteration methods. They are based on a decomposition of the system matrix A similar to

$$A = L + D + U \tag{2.15}$$

where L, D and U denote the strictly lower triangular, diagonal and strictly upper triangular parts of A. The algorithms differ in the formulation of the iteration step. For the Jacobi method it may be written as

$$D \cdot \Phi^{k+1} = -(L+U) \cdot \Phi^k + b. \tag{2.16}$$

whereas the Gauss-Seidel method is defined by

$$(D+L) \cdot \Phi^{k+1} = -U \cdot \Phi^k + b. \tag{2.17}$$

The stated systems have to be solved for each iteration in order to improve the current solution. For both methods, this generally requires little numerical effort as the inversion of D and $(D+L)$ respectively is inexpensive due to their simple structure. Even though the solution of the second version is slightly more complicated which results in an effort increased per iteration, it also reduces the overall number of iterations which generally makes it more effective.

2.3.2 Conjugate Gradients

The idea behind all gradient methods is the utilisation of the equivalence of the solution of (2.14) to the result of the substitution problem

$$\text{Minimise } F(\Phi) = \frac{1}{2}\Phi \cdot A\Phi - b \cdot \Phi. \tag{2.18}$$

For the purpose of minimising the function F, an initial guess u^0 is gradually improved by the successive minimization along a series of directions y^k according to

$$\text{Minimise } F(\Phi^k + \alpha y^k) \text{ subject to } \alpha \in \mathbb{R}. \tag{2.19}$$

As (2.18) defines a quadratic problem, the solution to 2.19 can be computed directly for positive definite matrices A which can generally be assumed in the context of continuum mechanics problems. With the help of the definition

$$r^k = b - A\Phi^k \tag{2.20}$$

the optimal value for α can be written as

$$\alpha^k = \frac{r^k \cdot r^k}{y^k \cdot Ay^k} \tag{2.21}$$

and the improved solution

$$\Phi^{k+1} = \Phi^k + \alpha^k y^k \tag{2.22}$$

follows.

The characteristic of the Conjugate Gradient (CG) method is the deduction of orthogonal search directions y^k which is simply achieved by means of

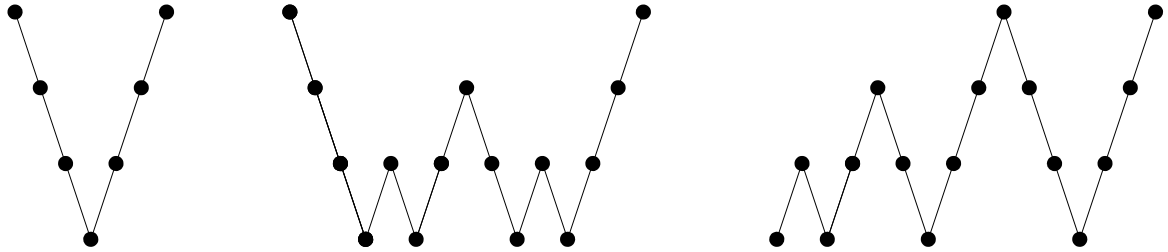
$$y^{k+1} = r^{k+1} + \frac{r^{k+1} \cdot r^{k+1}}{r^k \cdot r^k} y^k. \tag{2.23}$$

In theory, this approach leads to the exact solution of the original problem after n iterations at most (if n denotes the number of unknown components Φ_i) but in praxis, the algorithm is usually aborted when the residual r has fallen below a prescribed limit which greatly improves the numerical efficiency.

2.3.3 Multigrid methods

The term *multigrid methods* refers to a set of techniques generally reducing the required number of iterations for the solving of linear systems of equations with iterative solvers. As a result, they should not be considered as solvers but as a special class of preconditioning algorithms. They base on the idea that the exchange of information on finer grids is considerably slower than on coarse grids due to the local character of the FEM approach. Consequently, local deviations from the exact solution (often referred to as *high frequency errors*) are decreased much faster than (the usually more significant) global deviations (*low frequency errors*).

Multigrid methods try to improve the exchange of information by temporarily switching to modified problem formulations and performing a number of iterations there. The accordant steps are often referred to as follows:



(a) V-cycle (b) W-cycle (c) Full multigrid (FMG)

Figure 2.2: Typical courses of action for the transitions between finest (topmost circles) and coarsest (lowermost circles) grids during a multigrid run using four levels

Restriction: Transfer of (2.14) to a coarser grid formulation.

Smoothing: Reduction of the *high frequency errors* using a standard iterative algorithm like Gauss-Seidel. In this case, the term *high frequency errors* always refers to frequencies complying to the lattice spacing of the current grid.

Prolongation: Interpolation of the determined correction to the next finer grid.

These steps are generally executed repeatedly following predefined cycles as illustrated in figure 2.2. Here, a circle represents a number of smoothing steps on the correspondent grid level. A descent along a displayed conjunction between these circles stands for the application of a restriction operator while an ascension illustrates a prolongation. The exact construction of these operators depend on the applied type of multigrid method. *Geometric* versions use different discretisations of the problem domain representing a hierarchy of grids each with, for example, a doubled lattice spacing. On the other hand, *algebraic multigrid methods* only make use of information directly contained in the stiffness matrix.

During the course of this thesis only the generally more robust (but also slightly slower) algebraic type has been used and will therefore be introduced pursuant to [12]. Algebraic multigrid implementations always consist of two phases which usually are equally time consuming:

1. The *setup phase* in which the given problem is analysed concerning the suitable construction of coarse grids and the required grid transfer operators
2. The *solution phase* in which the actual grid transfers and smoothing steps take place

This process can best be described for a two-grid method which can directly be extrapolated to the desired number of grids. To this end, a linear system of equations in the fine grid (lattice spacig h) similar to

$$A^h \Phi^h = b^h \tag{2.24}$$

may be considered. The restriction operator I_h^H and the prolongation operator I_H^h (where the index H refers to the coarse grid) are determined during the setup phase while the simple relation

$$I_h^H = \left(I_H^h\right)^T \tag{2.25}$$

holds if A^h is symmetric. The task of assembling the prolongation operator I_H^h can be solved according to [12] and generally corresponds to the interpolation of the remaining error on the coarse grid to the fine grid. Finally, the *Galerkin operator*

$$A_H := I_h^H A_h I_H^h. \tag{2.26}$$

can be assembled which is used to carry out the grid transfers. This phase is followed by the actual solving process using multigrid cycles as introduced in figure 2.2 and the selected solver.

2.4 Parallelisation

The term *parallelisation* comprises a number of concepts allowing the simultaneous execution of instructions. This sector of computer sciences has been developing rapidly in recent years and has thus borne a complex theory on the topics of parallelisation techniques and the attainable speedup of program executions. However, only a general overview of accordant implementation concepts and evaluation standards is required for the understanding of this thesis and will therefore be provided in the following sections.

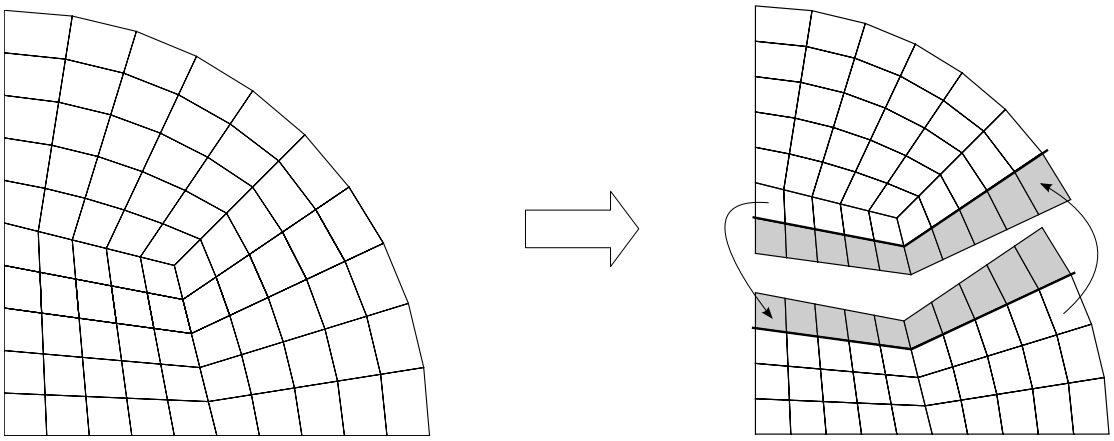


Figure 2.3: Meshed and partitioned problem geometry with ghost cells (grey) administering the exchange of information between adjacent partitions

2.4.1 Foundations of parallel computer systems

First of all, different hardware designs for parallel computing systems have been proposed and realised but only the following two have prevailed:

Shared memory: All processors have direct access to a common memory using a network. If all individual processors are identical, systems using this architecture are often also referred to as *symmetric multiprocessor systems* (SMP systems)

Distributed memory: Each processor possesses its own local memory while information required by other processors is exchanged by means of network communication

In addition to these plain views, mixed approaches exist of which clusters of SMP systems are most commonly used today. Here, identical groups of processors are designed as shared memory systems but these groups are interconnected via a network according to the definition of a distributed memory system.

Besides the hardware configuration, the programming of parallel computers is of particular importance. The applied programming models can generally be condensed under the heading of *message passing* which describes the data exchange between processors as the manual sending and receipt of messages containing the appropriate pieces of information. That way, the software implementation is independent from the underlying hardware architecture as a common interface to the message passing routines may be implemented according to the needs of shared memory and distributed memory systems in equal measure.

2.4.2 Parallelisation of the FEM

Concerning the parallelisation of computations making use of the FEM *grid partitioning* techniques are almost exclusively applied nowadays. The idea behind this approach is the spatial decomposition of a problem domain discretised according to chapter 2.2.2. The resulting *partitions* (or *subdomains*) are assigned to individual processors where they are treated like in the serial case. Of course, this formulation is still incomplete as the computation also requires information about (intermediary) results on adjacent partitions (that is, partitions sharing a common boundary with the active partition). This means that the solver regularly accesses data on the other side of subdomain interfaces.

For the sake of an efficient implementation of this event one layer auxiliary elements is introduced to the initially non-overlapping partitions and is placed along boundaries adjacent to other partitions. These *ghost cells* mirror the values of their counterparts on the related partition (See figure 2.3) by means of periodical updates. As a result, the communication effort and the required modifications of the solver are kept as small as possible.

2.4.3 Evaluation of parallel performance

For the rating of parallelised applications the *speed-up*

$$S_P = \frac{T_N}{T_P} \tag{2.27}$$

and the *efficiency*

$$E_P = \frac{N T_N}{P T_P} \tag{2.28}$$

are typically considered where P denotes the deployed number of processors, T_P the computing on P processors and N the number of processors used during the reference run. These performance indicators should be equal to $\frac{P}{N}$ and 100% respectively in the ideal case. Reasonable values for N might be 1 (standard case) or 2 (if solely the performance of the parallel algorithm on different numbers of processors is of note and no single-processor implementation is available).

In addition to these characteristic values based on exeuction times, the number of *floating point operations* or *FLOPs* (that is, the count of basic arithmetic operations on numbers with a mutable number of decimal places) often is of note in fields of numerical sciences. This is due to the fact that number of FLOPs generally is independent of the hardware implementation and thus algorithms may be compared irrespective of execution times. In these days, its value is often given in giga FLOPs (*GFLOPs*) which is more suitable in the context of the current performance of workstation computers. Another advantage is the possibility of splitting the overall efficiency into three distinct parts

$$E_P = E_P^{num} E_P^{par} E_P^{load} \tag{2.29}$$

by means of this definition. These mentioned portions stand for the *parallel efficiency*

$$E_P^{par} = \frac{CT(\text{parallel algorithm with N processors})}{P \cdot CT(\text{parallel algorithm with P processors})}, \tag{2.30}$$

the *numerical efficiency*

$$E_P^{num} = \frac{FLOPs(\text{best serial algorithm})}{FLOPs(\text{parallel algorithm on P processors})} \tag{2.31}$$

and the *load balancing efficiency*

$$E_P^{load} = \frac{CT(\text{one iteration on the full problem domain})}{P \cdot CT(\text{one iteration on the largest subdomain})} \tag{2.32}$$

if $CT(\cdot)$ marks the required computing time. Of these, E_P^{num} can be set to 100% if the applied algorithm remains unchanged which is usually the case for $N > 1$.

3 The ParFEAP software package

Now that the theoretical foundations of the FEM have been outlined in chapter 2 we can focus on a software implementing the emerging algorithms. FEAP 8.2 is such an implementation which is especially used in research and whose realisation details can be found in [17]. Since version 8.2, a special version of FEAP (called *ParFEAP*) can be built which has capabilities of using parallel solution algorithms. That way, solution times can be reduced dramatically on multiprocessor systems. This chapter deals with the structure, installation and usage of FEAP 8.2 in a Linux environment using the so called BASH[1]. FEAP 8.2 is generally compatible to Windows, UNIX, Linux and Max OS X based operating systems but the parallel functionality is currently only available on UNIX and Linux workstations and we will therefore restrict ourselves to these cases here. The differences using other shells are minor and will not be discussed here any further but, on the other hand, migrating to a UNIX-like environment can be quite challenging. As a result, this topic will be addressed in chapter 6 in detail.

3.1 Structure

3.1.1 Workflow of parallel computations

In the serial case, FEAP 8.2 is usually controlled by means of a single input file which contains the problem definition and the solution commands. Additionally, it is possible to submit commands to FEAP interactively (e.g. for creating plots). A parallel computation using ParFEAP however can generally be divided into three subsequent steps:

1. Subdividing the problem into a specified number of subproblems (*partition run*) which results in a separate input file for each partition

2. Running the program on the accordant number of processors utilising the input files generated in the first step (*parallel run*)

3. Performing an optional post-processing run on a single processor during which operations requiring global information can be performed (e.g. plots of the complete problem domain)

The whole process is non-trivial and makes use of several software packages which take care of the communication between the processes, provide the parallel solution algorithms etc. Figure 3.1 summarises the sequence of actions and should serve as an orientation guide during the reading of this chapter. In the following sections the concerned packages and their responsibilities will be introduced briefly.

3.1.2 Deployed packages and their roles

MPI

The MPI[2] itself is not a software package but a platform- and language-independent standard defining a communications protocol between processes during parallel program executions. Generally speaking, the participating processors exchange information simply by sending and receiving standardised messages.

There are several implementations of this standard available which allows the user to choose a software package according to his preferences. For the realisation of the validation tests (see chapter 4), the free software package MPICH2[3] was used because PETSc provides abilities to install this tool automatically which facilitates the integration. In contrast, the massive parallel environment tests made it necessary to use the so called IBM parallel environment. Accordingly, particularities evolving from this step will be addressed in chapter 6.1.

[1] Bourne-Again Shell – `http://www.gnu.org/software/bash/`
[2] Message Passing Interface – `http://www.mpi-forum.org`
[3] `http://www.mcs.anl.gov/research/projects/mpich2/`

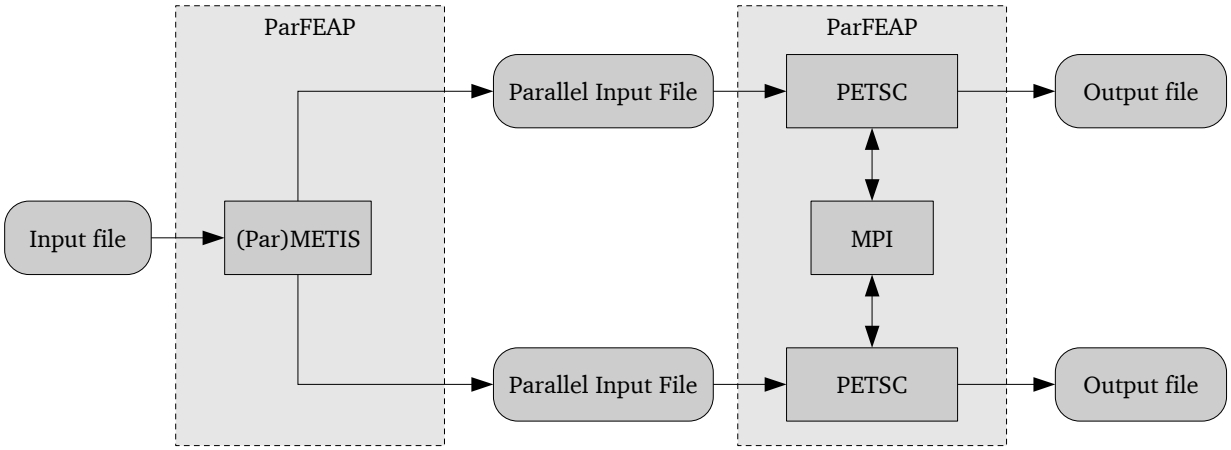


Figure 3.1: Simplified illustration of the workflow of a parallel computation with ParFEAP using two partitions

METIS

The family of software packages METIS[4] covers a wide variety of applications on graphs, meshes and matrices (Detailed information can be found in [6]) while only its ability of partitioning meshes is relevant here.

In our case, two packages based on the METIS libraries may be used: A serial version (simply called *METIS*) and the parallel, MPI-based version *ParMETIS*. Since ParMETIS includes the METIS libraries as well it is also capable of providing serial graph partitioning functionalities. As a result, we restrict ourselves to the more universal case of installing ParMETIS.

PETSc

PETSc[5] is a collection of mathematical algorithms with parallelisation support utilising MPI for the communication between processes. Due to its modular structure, a lot of extensions and especially numerical solvers have been implemented as PETSc routines. Unlike the traditional instance, a parallel built of FEAP 8.2 uses these algorithms instead of its built-in solver for the processing of linear systems of equations. Correspondingly, this fact has to be taken into account when evaluating timing results in chapter 6.2. An example of such a solver is *Prometheus* which will be introduced in the following section.

Prometheus

The optional parallel solver Prometheus[6] provides many solution algorithms associated with geometric and algebraic multigrid methods (see chapter 2.3.3). Being based on PETSc libraries it can be integrated in the solution process easily once PETSc has been installed. As a result, the potential speed-up of computation could be analysed in chapter 6.2.2 without much extra time and work on configuration issues.

BLAS/LAPACK

BLAS[7] and LAPACK[8] are software libraries providing basic and advanced functionalities for a great variety of numerical software tools. While BLAS is essentially dedicated to basic operations such as vector and matrix operations, LAPACK uses these definitions in order to implement solving strategies for eigenvalue problems, linear systems of equations, least-square problems and so on. Even though BLAS has become a de-facto standard within its range of application,

[4] `http://glaros.dtc.umn.edu/gkhome/views/metis`
[5] Portable, Extensible Toolkit for Scientific Computation – `http://www-unix.mcs.anl.gov/petsc`
[6] Scalable unstructured finite element solver – `http://www.columbia.edu/~ma2325/prometheus/`
[7] Basic Linear Algebra Subprograms – `http://www.netlib.org/blas/`
[8] Linear Algebra PACKage – `http://www.netlib.org/lapack/`

Software package	Version
FEAP	8.2
MPICH2	1.0.5-p4
ParMETIS	3.1
PETSc	2.3.3-p12
Prometheus	1.8.6
BLAS/LAPACK	3.1.1

Table 3.1: Deployed software packages and their versions

the implementation varies on different architectures due to the strong connection between the performance of advanced algorithms and the efficiency of the implementation of basic operations. Again, this may lead to deviations from the installation process described later in this chapter. An example of the consequences can be seen in chapter 6.1 where the IBM implementation ESSL has to be used in order to guarantee adequate performance.

3.2 Installation

The following instructions should provide a short summary of the steps that have to be performed for the purpose of creating an operational built of ParFEAP. Of course, this work cannot cover all special features of all of the presented software packages and hence only the standard installation will be mentioned. For additional information on the installation of the deployed packages, please refer to [15], [19], [2], [6], [7], [1] and the corresponding homepages.

Anyway, it should be noted that this section represents more than just a recapitulation of the instructions given in these manuals since a number of issues is undocumented (in particular, only a rudimentary installation manual for ParFEAP exists) and had to be resolved by hand.

Further on, the following sections assume that the user:

- already has basic knowledge in Linux and associated standard tools
- is using BASH (as mentioned earlier, this is not exactly necessary but other shells could make at least some of the instructions inapplicable)
- has a Fortran 77 and a C compiler installed on his system
- already obtained a license and the source code of FEAP 8.2
- has downloaded a current version of PETSc
- is using versions of the presented software packages which are compatible to the versions shown in table 3.1

3.2.1 Preparations

On the assumptions made in the previous section, the preparations mainly consists of the installation of PETSc. First of all, environment variables have to be set indicating the installation directories and the so called PETSc *architecture*. With architectures it is possible to manage multiple independent configurations of PETSc at the same time. This means, that it is for example possible to change from a version using debugging to another version without debugging by simply changing the *$PETSC_ARCH* environment variable. This avoids the needs of recompililing PETSc on each occassion. Furthermore, a list of predefined architectures exists (see *$PETSC_DIR/ config/*) featuring suitable predefined configuration options which can be individually adjusted.

In the case of a single user installation with debugging in a Linux environment, the commands for setting the environment variables could yield:

```
export PETSC_DIR=~/petsc-2.3.3-p12
export PETSC_ARCH=linux-gnu-c
export FEAPHOME8_2=~/feap82
```

Usually, these commands are placed in the file

```
~/.bashrc
```

which avoids the need to re-enter the commands for each new BASH-instance.

The next step is to extract the PETSc distribution to the folder defined by *$PETSC_DIR*. Afterwards, the package can be configured for example via

```
./config/configure.py --with-cc=gcc --with-fc=g77 --download-mpich=1 \
    --download-f-blas-lapack=1 --download-parmetis=1 --download-prometheus=1 --with-debugging=0
```

from the *$PETSC_DIR* directory. This command tells PETSc to use a C-Compiler named *gcc*, a Fortran compiler named *g77*, to download and configure the required software packages automatically (of course, this a requires an active Internet connection) and **not** to use debugging. In general, deactivation of the debugging features increases the performance by a factor of two or three but on the other hand, it also prevents PETSc from displaying numerous messages which might be useful during development. Consequently, it might be suitable to create a second configuration for PETSc (in this case *$PETSC_ARCH=linux-gnu-c-debug*). Switching between configurations may then simply be achieved by changing the accordant environment variable while this action has to take place before the compilation of ParFEAP.

If the recent action could be completed without errors, PETSc, MPICH2, BLAS, LAPACK and Prometheus should have already been configured properly (if not, please refer to [1]). Hence, the software package may be compiled and tested by issuing the command

```
make all test
```

which will also check for errors once the compilation has been completed. It should be mentioned that there might be some error messages concerning graphics if no running X-Server[9] could be found. Nevertheless, the installation should work properly if no other errors have been reported.

3.2.2 Creating a parallel version of FEAP 8.2

Having accomplished the preparation steps outlined in chapter 3.2.1, one can address FEAP and ParFEAP. First of all, this may require few modifications of *makefile.in* located in the *$FEAPHOME8_2* directory. The precise values that have to be set depend on the system settings and mostly define program names and compiler options. Given that the concerned system resembles a standard Linux installation, no changes have to be made here.

The compilation of the serial version of FEAP then can be started via

```
make install
```

and should be completed without errors. Afterwards the executable named *feap* should have been created in the *main* subdirectory and may be tested as desired.

Finally, an attempt can be made to build ParFEAP and the stand-alone parallel partitioner. This process is mainly undocument and requires manual fixing of current bugs in the installation process. ParFEAP depends on two packages, *ARPACK*[10] and *PARPACK*[11], which are included in FEAP's source files but have to be set up. In the *$FEAPHOME8_2/packages/arpack/archive* directory, the former can be installed by the same command which creates the library *arpacklib.a*. An identical procedure has to be conducted in the *$FEAPHOME8_2/parfeap/packages/arpack* directory but here, the file *makefile* contains a bug. This can be resolved by removing both occurrences of the term

```
*.F
```

(while the rest of the lines remains unchanged). The directory currently does not contain files applying to the given pattern which is why *make* would report an error during execution. Afterwards, the command

```
make install
```

may be performed in the same directory which (despite some warnings) results in the creation of a file called *parpacklib.a* in case of success.

Now that all libraries have been prepared, ParFEAP can be compiled which can be achieved by submitting the same command from the *$FEAPHOME8_2/parfeap* directory. In the case of using an old version of PETSc ($\leq$ 2.3.0), the file *usvole.F* in the same directory has to be modified prior to compilation in order to fix an incompatibility to the old PETSc API. Under such circumstances, please refer to the mentioned file and follow the instructions in the lines 231 to

[9] X Windows System Server - `http://www.x.org/`

[10] ARnoldi PACKage – `http://www.caam.rice.edu/software/ARPACK/`

[11] Parallel ARnoldi PACKage – `http://www.caam.rice.edu/~kristyn/parpack_home.html`

234. If all the demonstrated steps have been carried out correctly, an executable file *feap* ought to be present in the *$FEAPHOME8_2/parfeap* directory.

If desired, the stand-alone partitioner can now be addressed. It can be created in the *$FEAPHOME8_2/parfeap/partition* directory but once more, this makes modifications of the makefile necssary. Listing 3.1 shows the corrected version in which line 1 has been uncommented and the option *-lm* has been added in line 5 because the partitioner makes use of mathematical libraries which would be undefined otherwise. Using this corrected version of the makefile,

```
make partition
```

will result in another executable named *partition*.

```
1 include ${PETSC_DIR}/bmake/common/base
2
3 partition: partition.o sparse-graph.o
4     -${CLINKER} -o partition partition.o sparse-graph.o \
5         ${PARMETIS_LIB} ${MPI_LIB} -lm
```

Listing 3.1: Extract of the modified version of *$FEAPHOME8_2/parfeap/partition/makefile*

3.3 Running FEAP 8.2 on multiple processors

This section outlines the general use of ParFEAP while the focus is on the conversion of serial to parallel FEAP input files. To this end, the three required steps given in section 3.1.1 will be described in detail here while both versions of the partitioning step (using METIS or ParMETIS) will be addressed briefly after giving an overall introduction to FEAP input files.

3.3.1 Structure of a serial input file

As a starting a point, we refer to the example given in listing 3.2 which can be found in [14] in the original version. In this listing, the general structure of a basic serial input file can becomes apparent if one is familiar with the basic FEAP syntax which will be summarised here briefly:

- Each command name consists of four characters which are usually written in capital letters. The letters following such a command directly will be ignored and are only stated for improved readability
- All characters appearing behind an exclamation mark are be considered a comment and will thus be ignored
- After each multiline command a blank line (or a line containing comments only) is mandatory
- In general, command arguments follow a command separated by either a space or a comma (For an example of both valid versions, see lines 45 and 46). The version using a comma has the advantage that an optional parameter may be skipped by using two successive commas. For the exact syntax of each command, we again refer to [18]

We will now address the structure of the input file. The first two lines represent the header defining global settings like the number of dimensions or the number of degrees of freedom per element. The values corresponding to the number of elements may be set to zero in many cases which makes FEAP determine the counts itself.

The following section (lines 5 to 37) is called *mesh input data* section. Correspondingly, all settings concerning nodal coordinates, element coincidence, material models and boundary conditions can be found there. In this special case, an isotropic, linear elastic material model only permitting plane strain has been assumed. The command *END* (line 38) terminates this section.

The remaining part represent the *command language statements* which define the solution steps for the previously defined problem. The user can choose between a sequence of commands that will be carried out automatically (surrounded by the commands *BATCH* and *END*), an interactive prompt provided by FEAP or the combination of both features. For exampple, this makes the use of predefined solution commands in combination with interactive plot creation possible as, in general, commands available in batch mode are also available in interactive mode in similar form.

For an input file named *iex1serial*, the computation may finally be carried out by means of

```
/$FEAPHOME8_2/main/feap -iiex1serial
```

```
FEAP                        ! FEAP header
  9,4,1,2,2,4               ! 9 nodes, 4 elements, 1 material, 2 dimensional,
                            ! 2 degrees of freedom, 4 nodes per element

MATErial,1                  ! Material definition
  SOLId
  PLANe STRAin
  ELAS ISOT 1000.0 0.25     ! Linear elastic, isotropic material,
                            ! Young's modulus, Possion's ratio

COORdinates                 ! Nodal coordinates
  1 0 0.0 0.0
  2 0 4.0 0.0
  3 0 10.0 0.0
  4 0 0.0 4.5
  5 0 5.5 5.5
  6 0 10.0 5.0
  7 0 0.0 10.0
  8 0 4.2 10.0
  9 0 10.0 10.0

ELEMents                    ! Element coincidence
  1 1 1 1 2 5 4
  2 1 1 2 3 6 5
  3 1 1 4 5 8 7
  4 1 1 5 6 9 8

BOUNdary                    ! Boundary conditions
  1 0 1 1
  4 0 1 0
  7 0 1 0

FORCes                      ! Loads
  3 0 2.5 0.0
  6 0 5.0 0.0
  9 0 2.5 0.0

END                         ! End of mesh input data

BATCh                       ! Batch execution of solving commands
  ITERate                   ! User iterative solver
  FORM
  TANGent
  SOLVe
  DISP,ALL
  STRE ALL
END                         ! End of solution command statements

INTEractive

STOP                        ! End of input file
```

Listing 3.2: Standard serial input file (*iex1serial*) for FEAP 8.2 (Adapted from [14])

from the directory where the serial input file is located. The displayed command line argument is in fact optional and may also be omitted. In this case, FEAP will prompt for a filename which will be of further importance in the parallel case (see chapter 3.3.3). During a serial run, FEAP will create/overwrite several files of which the most important ones have the following purposes:

Oex1serial: The main output file containing all the results (in this case, e.g. the nodal stresses and displacements according to the lines 47 and 48 of the input file) and a history of the executed commands and the respective timings

Lex1serial: An additional log file providing a convenient summary of the progress of solver iterations (if an iterative solver has been chosen) and their timings

Rex1serial: An optional file making the user able to restart a computation at a restart point created via *EXIT*

feapname: Internal file which holds the name of each of the input and output filenames valid for the recent run

3.3.2 Partitioning

Now that the basic structure of a serial input file has become apparent, the necessary adjustments for the use of ParFEAP can be approached. Regardless of which graph partitioning technique is favoured (METIS or ParMETIS), the serial input file has to be modified slightly though in a different way. In any case, the solution commands have to be replaced by commands controlling the partitioning process. The particular commands depend on the chosen partitioning technique and will be addressed in the accordant subsections.

The solution commands extracted from the serial input file have to be moved to another file. According to the FEAP naming specifications, this file has to be named *solve.filename* if the original input file is named *ifilename* (as the name of every input file should start with the letter "i" according to FEAP's naming conventions). The result of this procedure can be seen in listing 3.3. If the naming convention has been followed, each parallel input file (see the following section) should automatically include this file.

```
1  BATCh                ! Batch execution of solving commands
2    PETSc,ON           ! Use PETSc (optional, should be activated by default)
3  ! ITER               ! Never (!) use the ITER-command with this syntax in parallel
4                       ! mode as FEAP will crash that way. Instead, either omit the
5                       ! command (The solver will be chosen via command line
6                       ! parameters) or specify the solution tolerances explicitly
7    FORM
8    TANG
9    SOLVe
10   DISP,ALL
11   STRE ALL
12 END
```

Listing 3.3: Solution command input file for a problem based on figure 3.2 (*solve.ex1parallel*)

This step concludes the necessary changes and eventually allows the execution using ParFEAP. Of course, ParFEAP defines several new commands and additional parameters for existing commands but besides the changes concerning the *ITER* command stated in figure 3.3 there is generally no necessity to adjust the solving commands. However, this is not true for plot commands which is why this topic will be addressed extensively in the chapters 3.3.4 and (making use of the possibility to incorporate custom macros into FEAP) 5.2.

Finally, the preparations have been completed and the partitioning can be carried out. As mentioned before, this can be done using a single or multiple processors. For the second alternative, it is essential that the stand-alone version of the partitioner has been created successfully (see chapter 3.2.2 for instructions).

METIS version

Listing 3.4 shows that the original solution command statements have been replaced by the the commands *GRAP* and *OUTD* (lines 10 and 11) which is the standard syntax for the application of METIS. These commands control the creation of the input files for the parallel run (see figure 3.1). In this case, the first statement makes FEAP use METIS in order to

create two partitions. Of course, the entered number of domains should equal the desired number of processes for the parallel run. Finally, the *OUTD* commands instructs FEAP to write the parallel input files.

```
1  FEAP
2    9,4,1,2,2,4
3
4  {...}
5
6  END                  ! End of mesh input data
7
8  BATCh                ! Batch execution of solving commands
9    GRAPh,,2           ! Create two partitions
10   OUTDomains         ! Write input files for parallel solution
11 END                  ! End of solution command statements
12
13 STOP
```

Listing 3.4: Parallel mesh input file (*iex1parallel*) for a problem based on figure 3.2 using 2 processes and METIS

With an input file modified according to the above instructions, the partitioning step can be started via

```
/$FEAPHOME8_2/parfeap/feap -iiex1parallel
```

which, in this case, will create two new files named *iex1parallel_0001* and *iex1parallel_0002*. Additionally, a file called *solve.ex1parallel* will be created if it does not already exist.

Statistics on the results of the partitioning process may be extracted from console output. They include information about the total number of elements, the number of ghost cells and the number of nodes and elements per partition.

ParMETIS version

For the use of ParMETIS, two possible approaches exist. The first approach consists of launching the *partition* executable via MPI by hand while the second one utilises a shortcut making ParFEAP perform the appropriate call itself. Of course, the manual version is less convenient and, even more important, generally considerably slower since the mesh data has to be read completely for each separate run. On the other hand, the second approach is experimental and usually makes some adjustments of the source code necessary. For the above reason and for the sake of a better understanding of the underlying processes, we will focus on the first course of action here and refer to appendix B for the latter.

First of all, a "flat" version of the input file has to be created. This can be achieved through a run of ParFEAP with the parallel input file but with *OUTM* as the only (!) solution command. That way, ParFEAP will create a new input file with the extension *.rev* (For the featured example, this would yield *iex1parallel.rev*). Afterwards, the partitioning process may be started as an MPI run via

```
/$PETSC_DIR/externalpackages/mpich2-1.0.5p4/mpiexec -n nump \
    /$FEAPHOME8_2/parfeap/partition/partition numd iex1parallel.rev
```

where *nump* denotes the desired number of processes and *numd* represents the number of partitions that should be created.

In case of success, a file named *graph.ex1parallel.rev* should have been created. This file already contains the partition information but still has to be transformed into the appropriate number of parallel input files. To this end, another ParFEAP run is necessary, either by using a modified version of *iex1parallel.rev* as input file (see figure 3.5) or by making use of the interactive mode. In our example, this makes the call

```
/$FEAPHOME8_2/parfeap/feap -iiex1parallel.rev
```

necessary which should lead to a result similar to the METIS version.

3.3.3 Solving

In order to initiate the parallel solution process, ParFEAP has to be started through MPI with a number of processes greater than one because ParFEAP will not initialise the PETSc interface otherwise. So, again referring to our example, the execution could be started with

```
FEAP
  9,4,1,2,2,4

{...}

END                  ! End of mesh input data

BATCh
  GRAPh,FILE         ! Use partition information based in file input
  OUTDomains         ! Write input files for parallel solution
END                  ! End of solution command statements

STOP
```

Listing 3.5: Modified solution command part of a file created from 3.4 using the *OUTM* command

Parameter	Purpose	Common settings
-ksp_type	Choses the top level solver	*cg* (Conjugate gradients)
-ksp_rtol	Relative resdiual tolerance	*1e-08*
-ksp_atol	Absolute resdiual tolerance	*1e-16*
-pc_type	Choses the preconditioner	*jacobi* *prometheus*
-pc_mg_type	Defines the type of multigrid cycles	*multiplicative* (Standard cycles) *full* (FMG)
-log_summary	Monitor code performance without distorting timing results	
-get_total_flops	Outputs the total number of FLOPs over all processors	
-options_left	Displays a list of command line parameters which were **not** used during the run (useful for finding misspelled parameters)	
-ksp_view	Shows which solvers are being used for each iteration	
-ksp_monitor	Enables verbose residual outputs for each iteration. Should **not** be used for timing issues	

Table 3.2: Common PETSc command line parameters

```
/$PETSC_DIR/externalpackages/mpich2-1.0.5p4/mpiexec -n 2 \
    /$FEAPHOME8_2/parfeap/feap options
```

where *options* denotes a sequence of command line parameters that will be passed to PETSc directly. Consequently, the solver may solely be controlled by means of these parameters. Table 3.2 gives a short introduction to some of the most important options. For the complete list, please refer to [1]. Once the program instances have been started, ParFEAP will prompt for a single filename even though multiple files are required. For that reason, the first of the parallel input files has be specified (in this case *iex1parallel_0001*) while ParFEAP tries to determine the subsequent input files automatically.

When all processes have been assigned an input file, each one begins with the execution of commands located in *solve.ex1parallel*. This happens while only exchanging MPI messages concerning ghost cells. As a result, all values are only stored locally during a parallel run which also has a special consequence for the creation of plots: It is not possible to create a plot of the whole problem domain during a parallel run but each process will rather create a plot of its assigned partition. Nevertheless, it is possible to collect global information by means of special instructions. A subset of the related commands, options and their effects can be taken from table 3.3 while the complete list is available in [19].

During each parallel run ParFEAP creates a number of output files per process. Each of these files is an equivalent to the respective serial file (e.g. *Oex1parallel_0001*), but naturally only contains local information.

3.3.4 Post-processing

Post-processing plays an important role in terms of the FEM. In our context, the post-processing procedure depends on the kind of information that is required. Whenever local considerations are sufficient for a specific problem there is no need to change the post-processing procedure suitable for serial computations which is why this topic will not addressed

Command	Purpose
GLISt,,n	Define the list with id *n* of up to 100 nodes (where n may range from 1 to 3). The global node numbers of nodes that belonging to the list have to be entered after the END command
DISP,LIST,n	Creates one file per process containing the displacements associated with global node numbers of the *n*-th list of nodes defined with *GLIS*. Also works for *STREss*, *VELOcity* and *ACCEleration*
DISP,GNODe,start,end,n	Similar to *DISP,LIST*, but without the need to define a list. Outputs every *n*-th nodal displacement/velocity/... of nodes with global numbers from *start* to *end*.
GPLOt,DISP,n	Writes one file per processor containing the *n*-th component of the nodal displacements associated with global node numbers of all nodes. Works with *STREss*, *VELOcity* and *ACCEleration*
PLOT,NDATa,DISP,n	Plots the contents of a file created via *GPLO*

Table 3.3: ParFEAP in the context of global node numbers

here. In contrast, it is often easier or more convenient to evaluate results from a global perspective. In cases where global node numbers are essential, this makes the usage of the commands listed in table 3.3 inevitable. Of these commands, *GPLO* and the plot command *NDAT* play a special role as they allow the creation of plots of the global problem domain. The procedure has to be prepared by creating several files using *GPLO* which may be processed afterwards. For these purposes, another single processor run of ParFEAP in the **interactive** mode is necessary. That way, the *NDAT* command instructs ParFEAP to look for a global plot file containing the required information. If found, the content of this file is read and directly forwarded to an active plot.

As an example, one could want to create a plot of the nodal displacements in x direction. In the parallel run, this would be prepared by means of

```
GPLO,DISP,1
```

which would create one file per partition (for each time step!) with the appropriate contents. Once the parallel run is finished, a plot could eventually be created via

```
PLOT,NDAT,DISP,1
```

and then be manipulated as usual.

Lastly, it should be mentioned that alternative visualisation technique exist. Chapter 5.2 presents an example of a custom way of bypassing the drawbacks of the original plot interface.

4 Validation

The ParFEAP software package introduced in chapter 3 is quite new and relies on several other packages. Therefore, it is generally favourable to validate the produced results using sample problems which utilise different program features. This way, also certain problems concerning some setting combinations could be revealed.

This chapter outlines the analysed problems and the numerical settings which lead to the encountered results. Additionally, the applied tests will be described and a subset of the obtained results will be displayed. For the sake of clarity and an easier reading, the source of the relevant FEAP input files can be found in appendix A. In all cases, it can be expected that the results obtained during a parallel run should equal the serial equivalents up to terms of numerical precision. This expectation has been checked by means of three simple considerations:

1. Are the computed maximal principal stresses identical? Especially, is this true for each time-step?
2. Are the computed stresses/displacements of randomly chosen test nodes identical in both cases?
3. Are boundary and contact constraints maintained with comparable tolerances?

Finally, it should be noted that the presented test cases base on original FEAP input files provided by the supervisor and have been modified by the author.

4.1 Numerical setup

The applicable numerical settings have been kept unchanged for the chosen validation tests. They can be extracted from table 4.1 and rerpresent ParFEAP's default settings in most cases.

Option	Setting	Affected Tests
Serial top level solver	Conjugate gradients (FEAP internal)	All
Parallel top level solver	Conjugate gradients (PETSc)	All
Standard preconditioner	Jacobi	All
Multigrid preconditioner	Prometheus	All
Multigrid smoother	Gauss-Seidel	All
Multigrid cycles	Standard V-cycles	All
Relative residual tolerance	$1.0 \cdot 10^{-8}$	All
Absolute residual tolerance	$1.0 \cdot 10^{-16}$	All
Maximum augment iterations	5	2nd, 3rd
Maximum Newton iterations	30	2nd, 3rd
Contact interaction	Frictionless	3rd
Contact constraint penalty	$1.0 \cdot 10^{3}$	3rd
Initial penetration check tolerance	$1.0 \cdot 10^{-5}$	3rd
Maximum gap considered as contact	$1.0 \cdot 10^{-5}$	3rd
Out of segment contact tolerance	$1.0 \cdot 10^{-5}$	3rd

Table 4.1: Summary of numerical settings

The versions used to create the ParFEAP built correspond to the versions displayed in table 3.1, while PETSc was configured without debugging.

4.2 Linear-elastic problem

The first test consists of a pierced linear-elastic disk under a static, uniaxial load as shown in figure 4.1. The underlying mesh is composed of unstructured tetrahedral elements and was available in four versions featuring 3523, 13738, 33214 and 147987 elements. The body's upper surface is clamped while the lower surface is being pulled downward with a constant force of 50 N.

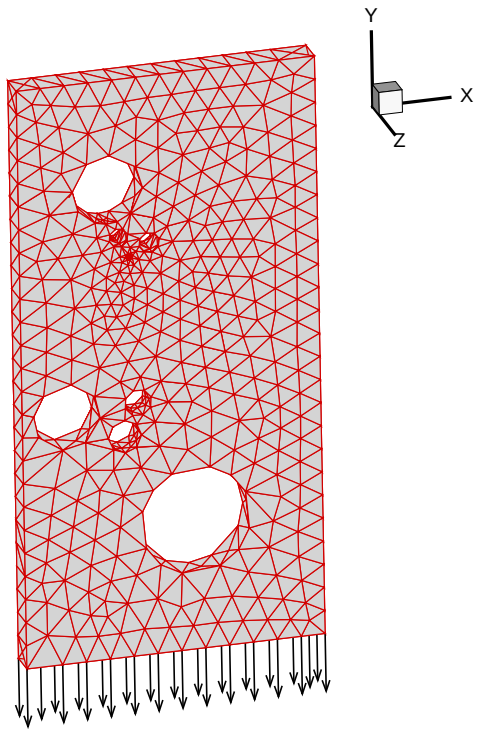
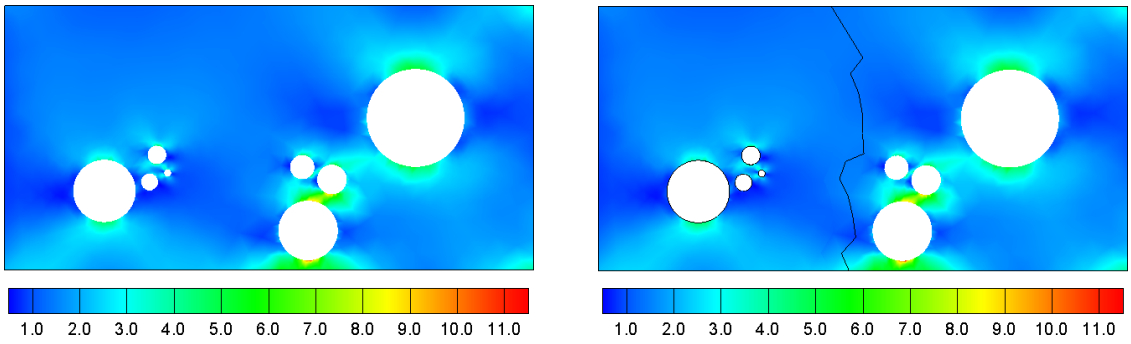


Figure 4.1: Meshed problem geometry (coarsest grid) of the first and second test problem

The acquired results were in all cases encouraging as the encountered stresses and displacements where identical throughout this test. This fact is exemplified by figure 4.2 which illustrates the calculated von Mises stresses for the serial and the parallel run. In addition to the displayed visual evidence, direct comparisons of the computed values have shown the same result. A subset of these findings has been condensed to table 4.2.

4.3 Transient plastic problem

The second test case is an extension of the first validation problem. The mesh itself remained unchanged and hence figure 4.1 describes the problem geometry for this case as well. This is also true for the boundary condition on the upper surface though a prescribed displacement has gradually been applied to the lower part this time. More precisely, the displacement is linearly increased to 2% of the length of the plate within half a second. As a result, the formulation has been transformed into a transient (but quasi-static) problem using a time-step size of 0.01 seconds. The material model this time included covered plasticity effects and has been tested for small and finite deformations in equal measure. In this context, the uniaxial yield stress has been set to $450\,\mathrm{N/mm^2}$.

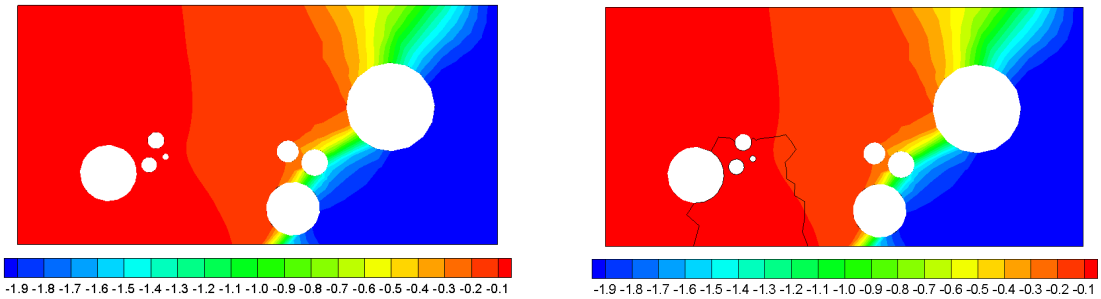


(a) Serial case (b) Parallel case using two partitions

Figure 4.2: Contour plot of the computed Von Mises stresses in $\mathrm{N/mm^2}$ for the first validation test on the finest mesh

Node number	Coordinates x	y	z	Displacement in y direction serial	parallel
1	0.0000	0.0000	0.0000	-0.00272610	-0.00272610
3000	38.2259	139.4584	10.0000	-0.00062057	-0.00062057
6000	35.7344	139.1676	3.7038	-0.00064460	-0.00064460
9000	44.2922	145.8013	1.4231	-0.00045069	-0.00045069
12000	38.4604	137.5516	1.8246	-0.00064523	-0.00064523
15000	40.7328	138.5438	9.5638	-0.00064157	-0.00064157
18000	32.2390	137.3185	9.5090	-0.00068635	-0.00068635
21000	37.5605	140.1476	9.6579	-0.00061059	-0.00061059
24000	36.8743	139.9469	1.2808	-0.00061303	-0.00061303
27000	35.7996	137.1891	8.0913	-0.00068013	-0.00068013
30000	35.5292	137.4570	6.4297	-0.00067663	-0.00067663

Table 4.2: Juxtaposition of displacements for the first test case computed during a serial on the finest mesh



(a) Serial case (b) Parallel case using two partitions

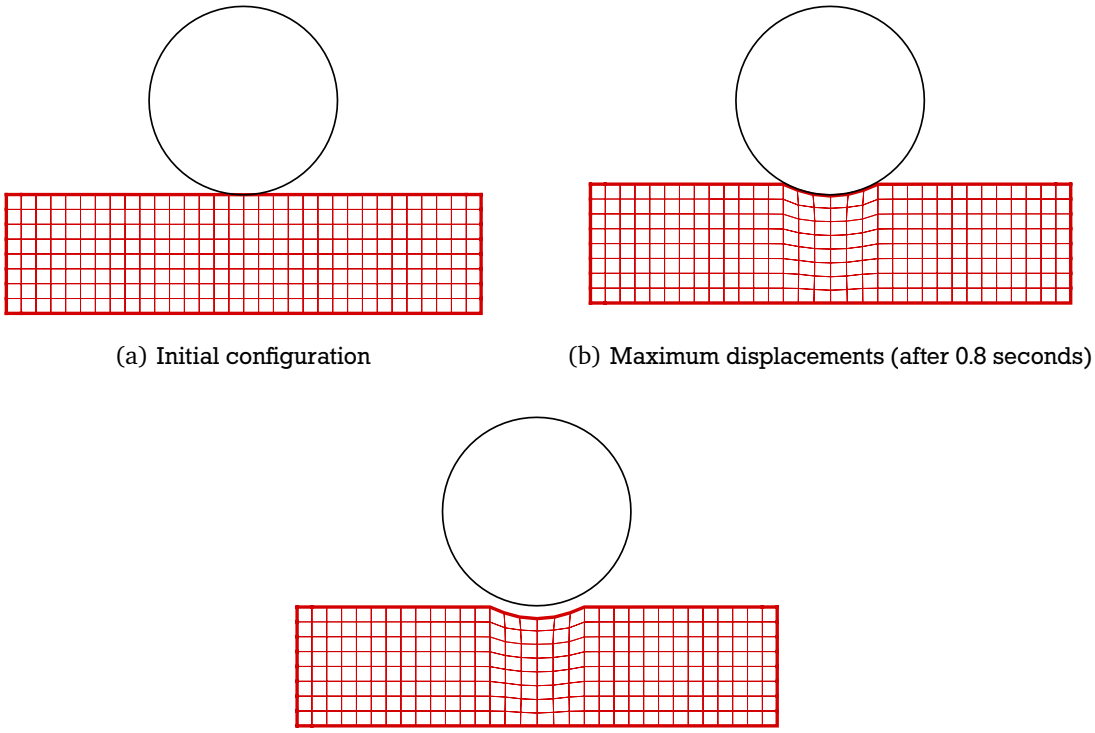
Figure 4.3: Final y displacements for the second validation test on the finest mesh

Examinations similar to the comparisons in table 4.2 for this configuration using a small displacement formulation once more showed indistinguishable results for the serial and the parallel solver. A respective example is figure 4.3 which illustrates the computed displacements for the final time step. When however a finite displacement model was applied, no results could be obtained because the respective runs always aborted due to an invalid residual norm. Unfortunately, this flaw could not be resolved in the course of this thesis.

4.4 Plastic contact problem

ParFEAP's capabilities concerning contact functionalities where subject to the last validation test. More precisely, the contact of an elasto-plastic block with a rigid cylinder using a structured grid and a mixed strain formulation (see [17]) illustrated in figure 4.4 has been considered (For more information on the topic of mixed element formulations in FEAP/ParFEAP refer to [17], pages 45ff). The whole problem is transient and takes one second during which the block is moved towards the rigid body (in FEAP, only meshed geometries can be moved which is not true for the cylinder) and returned after the maximum displacement has been reached after 0.8 seconds. To this end, a prescribed displacement is applied to the lower boundary forcing a maximum penetration of 1 mm of the block which is 10 mm a thick . As stated in table 4.1, the cylinder with a radius of 8 mm is considered frictionless.

For the second time, this configuration revealed and an undocumented incompatibility in the parallel built. The problem definition did not lead to any errors during the execution but the examination of the extracted data sets showed that the contact definitions had been ignored by ParFEAP. This means that the meshed block had been moved but not deformed at all. In order to rule out a false definition of the input file, several versions of the format stated in appendix A.3 have been tested but the expected results could not be achieved in any case. As a result, it can be concluded that ParFEAP is currently not applicable for contact problems using a rigid contact surface.



(a) Initial configuration

(b) Maximum displacements (after 0.8 seconds)

(c) Final configuration (after one second)

Figure 4.4: Illustration of the configuration and the serial results for the third validation problem

4.5 Conclusion

All in all, no inconsistencies between the results using the serial and the parallel built of FEAP could be found at all which implies the tool is ready for the productive use with standard problems. Of course, this conclusion cannot be extended to all features which has been shown by means of the scond and third validation test. In terms of advanced features offered by FEAP this chapter could only cover cover a small subset of the available possibilities and that is why additional tests are recommended before making use of them. On the other hand, a wide variety of tests has already been applied to the standard features of the software. For example, more validation tests and results can be found in the appendix of [19], but in this case without documentation of the deployed input files and the numerical settings. In this context, the preceding considerations can be seen as a verification of the results presented in the original manual.

5 ParFEAP tool programming

Both preceding chapters have been dedicated to ParFEAP's features and their usage in a workstation environment. But besides the apparent advantages of a solving mechanism which supports parallelisation, it is also necessary that this feature can be employed efficiently. As a result, this chapter deals with possible simplifications of the productive application of ParFEAP.

In chapter 3.3.3 it has already been mentioned that ParFEAP (in contrast to FEAP) offers no built-in batch-processing support. While this may be acceptable on workstations, environments preventing interactive program control exist as well. For example, this will be the case in chapter 6 where a so called *load leveler* makes batch execution mandatory. That is why the first part of this chapter is dedicated to the use of so called shell-scripts in order to circumvent this drawback in a Linux/UNIX environment.

But scripts are not the only way of influencing the interaction with ParFEAP as the software package is shipped with the application's complete source code. In addition to the obvious advantage of being able to inspect the implementation of code sections (and maybe to fix bugs by oneself in urgent cases), it also enables the user to customise a ParFEAP built. While the modification of FEAP's core may not recommended for most users, FEAP (and hence ParFEAP) offers the possibility to define custom commands by means of so called "user macros". An respective example adding an interface to the plot tool Tecplot®[1] will therefore be presented in the second part of this chapter.

5.1 Batch processing of ParFEAP jobs

Shell scripts are one the most important tools for the work in a Linux or UNIX environment, especially for the batch execution of a sequence of commands. For this purpose, a variety of scripting languages with different advantages and disadvantages exist. In this context, the KornShell[2] has been chosen due its widespread usage and support on many systems while, in this case, especially the support on the *IBM AIX* system used in chapter 6 was of particular importance.

During the course of this thesis, a script has gradually been developed offering a list of convenient features for the batch execution. For the sake of clarity, only the core aspects and ideas will be described here. The script itself including a small documentation can be found in appendix D and offers the following features:

- Allows batch execution of parallel runs
- Permits an optional partitioning run preceding a parallel run
- Checks whether all parallel input files are accessable which facilitates debugging
- Offers convenient control via command lines parameters
- Collects important files corresponding to the current in a single archive for an improved traceability

The main task certainly is to bypass the otherwise mandatory user interaction for the specification of the name if the input file. For these purposes, one takes advantage of a Linux/UNIX feature which allows the redirection of the standard input. This means, that input which is usually entered by the user can be read from a file or similar sources. This leads to a syntax analogue to listing 5.1 if *$MPIEXEC* and *$PARFEAP* represent the path to the MPI and ParFEAP executables, *$nproc* denotes the desired number of processors and *$filename* holds the name of the first parallel input file. What happens here is that the content between both appearances of *EOT* will be interpreted as if it was the content of a file which serves as user input. As a result, the stated code segment sends five commands (lines 10 to 14) to ParFEAP once it has been initiated via MPI: The file name of the first input file, four empty lines (interpreted as keystroke of the return key) and the letter "y". To ParFEAP, this means that a filename is supplied, the proposed names of the four subsequent files are confirmed and the question whether all inputs where correct is answered with "y" for yes. The previous deletion of the file *feapname* (line 7) ensures that this sequence of commands is always valid provided that the file referenced with *$filename* exists and is readable. This may for example be ensured by a preceding check which can be performed according to lines 1 to 5.

In addition to this necessary feature, one can think of numerous repetitive tasks which can be handled by a script automatically. For example, a basic backup feature has been implemented which places all files belonging to a ParFEAP

[1] `http://www.tecplot.com`

[2] KornShell Command And Programming Language – `http://www.kornshell.com/`

```
1  if [ ! -e $filename ]
2  then
3    echo 'Given file does not exist or could not be read'
4    exit 1
5  fi
6
7  `rm feapname`
8  $MPIEXEC $PARFEAP -procs $nproc <<EOT
9  $filename
10
11
12
13
14 y
15 EOT
```

Listing 5.1: Code segment changing the source of the user input from keyboard to a predefined set of commands

run in a single archive. This prevents subsequent runs from overwriting the current results while the archive's filename contains the timestamp of the execution in order to keep track of the solution history. Listing 5.2 shows an accordant implementation also supporting the additional backup of the input files so that results can be matched with the associated definitions.

The first part of the code fragment (lines 1 to 25) defines a function which may be used to assemble the suffix of files created during a parallel run. For example, the suffix equals *0001* for the first file and *0032* for the thirty-second file of a specific file type. For this sample, i-, L- and O-files have been incorporated (lines 31ff) and are being added to an archive created in line 29. Of course, this fragment can be extended for other, optional files (e.g. restart files or plots) easily and, consequently, this has been done in the complete script stated in appendix D.

5.2 Simplifying the visualisation for parallel runs

There are several handicaps concerning ParFEAP's (and FEAP's) plotting capabilities which evoke the demand for an alternative for special purposes. First of all, this refers to the limitation of disjointed plots for all partitions during parallel runs. As mentioned before, this may be circumvented via an additional serial run though might not be desired in all cases.

On the other hand, FEAP's plot interface itself possesses some fundamental drawbacks in terms of interactive manipulation of the display. This lead to the idea of creating an interface to the plot tool Tecplot® . The required functionality for the serial version of FEAP has been realised by the supervisor by means of a FEAP user macro. As a result, the presented expansion to the parallel case may be used to demonstrate the possibilities offered by user macros at the same time.

In fact, FEAP offers more customisation possibilities than this introduction can cover. Macros may for example define custom mesh functions, material models, plot commands or element types besides the presented solution command functions which are subject so this chapter. For an outright coverage of this topic, please refer to [16].

5.2.1 Existing groundwork

As indicated before, the presented implementation of the Tecplot® interface depends on a FEAP user macro written by the supervisor and hence the structure of this macro will be outlined here using small code fragments. For the complete source code including changes by the author, please refer to appendix C. FEAP user macro's are placed in *$FEAPHOME8_2/user* in the files *umacr0.f* to *umacr9.f* and have to be written in Fortran or C. In the original form, these files contain a skeleton of such a macro. Listing 5.3 shows this exemplified by the file *umacr1.f* while omitting some of the included comments.

For the purpose of defining a custom solution command, line 19 has to be uncommented and the word *name* has to be replaced by the desired unique identifier of the custom command. The rules for a valid command name are those which apply to all FEAP commands and, as a result, the name has to consist of four significant characters. In the case of the TECP interface subject to this chapter, the value of *uct* has been changed to *tecp*.

Naturally, a custom command usually consumes numerical and/or textual arguments just like a built-in command does. This functionality is provided by the parameters *lct* for strings and *ctl* for up to three real numbers. To elucidate

```
getSuffix ()
{
  id=$1

  infix=''
  case $id in
    [1-9])
      infix='000'
      ;;
    [1-9][0-9])
      infix='00'
      ;;
    [1-9][0-9][0-9])
      infix='0'
      ;;
    [1-9][0-9][0-9][0-9])
      ;;
    *)
      echo 'Invalid file id'
      exit 5
      ;;
  esac

  echo "${infix}${id}"
}

now=`date +%s`
archivename="run${now}.tar"
`tar -cf $tarfilename "solve.${filename}"`

for filetype in i L O
do
  serialfile="${filetype}${filename}"
  `tar -uf $archivename $serialfile`

  i=1
  while [ $i -le $nproc ]
  do
    parfile="${serialfile}_`getSuffix ${i}`"
    `tar -uf $archivename $parfile`
    (( i += 1 ))
  done
done
```

Listing 5.2: Example implementation of an automatic backup tool using a shell-script

the usage of these parameters, we refer to the sample of usage of the Tecplot® interface given in listing 5.4. In this case, lines 3, 7 and 8 contain the calls the macro should be able to understand. Correspondingly, *lct* will be equal to *init*, *write* or *close* while no value has been supplied for *ctl*. Consequently, the user macro can distinguish the requested action from the value of *lct* by means of a code fragment similar to listing 5.5. For an operational built, these lines would be incorporated into listing 5.3 around line 25. The two remaining cases of the predefined if-statement in listing 5.3 correspond to restart-functionality and only have to be changed if special actions have to be performed in this case.

Now that the main structure of a user macro has become clear, the desired actions may be implemented. Generally, this requires access to global variables like the number of nodes, the number of elements, the current time-step and/or corresponding values (e.g. nodal displacements). FEAP provides access to this data via a list of common header files located in *$FEAPHOME8_2/include*. Accordingly, access to specific variables may be gained by including the appropriate header file. Table 5.1 lists a subset hereof providing access to commonly required variables.

```
1       subroutine umacr1(lct,ctl,prt)
2
3       {...}
4
5       implicit none
6
7       include 'iofile.h'
8       include 'umac1.h'
9
10      logical  pcomp,prt
11      character lct*15
12      real*8    ctl(3)
13
14      save
15
16c     Set command word
17
18      if(pcomp(uct,'mac1',4)) then      ! Usual form
19c       uct = 'name'                    ! Specify 'name'
20      elseif(urest.eq.1) then           ! Read restart data
21
22      elseif(urest.eq.2) then           ! Write restart data
23
24      else                              ! Perform user operation
25
26      endif
27
28      end
```

Listing 5.3: Header of the considered user macro

The original macro creates a new file called *tecoutXXX.dat* where *XXX* represents the file's consecutive number. This file contains the nodal coordinates, displacements, updated positions (that is, the nodal coordinate in the deformed mesh), forces, the three principal stresses and the three invariants of the stress tensor of all nodes (The third invariant corresponds to the Von Mises stress which is relevant for most cases). As a result, the respective files usually consume substantial memory and that is why such plots should generally not be generated for each time-step of large problems.

Finally, the obtained results may be imported into Tecplot® by means of "File -> Load Data File(s)" using the "Tecplot Data Loader" import format. Once the file has been loaded, arbitrary plots can be created making use of the full functionality of Tecplot® which includes mesh, contour and vector plots, interactive plot manipulation, animation and much more. For a detailed introduction to the available possibilities, refer to [20].

Filename	Variable name	Matter
comblk.h	hr	Array containing real-valued information like stresses and displacements
	mr	Array holding integer values like node numbers
counts.h	nstep	Number of processed time steps
tdata.h	dt	Time step size
pointer.h	numnp	Number of nodes in the current partition
	numel	Number of elements in the current partition including ghost elements
pfeapb.h	numpn	Number of nodes in the current partition excluding ghost nodes
setups.h	processor	Total number of processors

Table 5.1: Important header files for the application in FEAP user macros

```
1  BATCh
2    LOOP,time,10
3      TECP,init            ! Open a new output file
4      TIME
5      TANG,,1
6      STRE,NODE            ! Compute nodal stresses (Mandatory!)
7      TECP,write           ! Write post-processing data to the file opened before
8      TECP,close           ! Close file
9    NEXT
10 END
```

Listing 5.4: Example of use of the Tecplot® user macro

```
1        if(pcomp(lct,'init',4)) then
2  c     'TECP,init' -> Open a new file
3        elseif(pcomp(lct,'close',4)) then
4  c     'TECP,close' -> Close currently active files
5        elseif(pcomp(lct,'write',4)) then
6  c     'TECP,write' -> Write current values to the active file
7        else
8  c     Unknown option -> Display error message
9        endif
```

Listing 5.5: Identification of the requested action in the shell script

5.2.2 Extension to the parallel case

The Tecplot® macro was originally designed for the use with a serial built of FEAP and hence it does not support parallel runs. To go into details, the implemented file creation is unsuitable for the parallel case as each process will try to access the same file which inevitably corrupts its contents. This flaw is often referred to as *race conditions* in the context of computer software. The term implicates that the result of the execution depends on the timing of the file access by different processes which may be different for each run. Additionally, each process still has access to local node information only and is not able to create a global Tecplot® output file. Lastly it has to be mentioned that the original implementation naturally does not distinguish between ghost and substantial nodes (and consequently elements).

Of these issues, solely the file corruption problem necessitates significant modifications. This is true due to the fact that Tecplot® requires neither global node nor global element information. Instead, multiple data files containing information related to independent partitions may be loaded simultaneously which offers the possibility of creating one output file per process. Furthermore, the distinction between ghost and substantial items is not strictly necessary (and hardly possible because of ParFEAP's internal structure) as processes always store values for linked cells. This naturally leads to the fact that plots created using this macro will always contain redundant data to some extent. On the other hand, this effect will not be visible at all for plots of the whole problem domain. Even for partition plots this only limits the possibility to show the *exact* shape of the problem domain but does not distort the obtained results at all. Accordingly, it has been decided that the drawback of duplicate ghost cell information does not outweigh the high implementation complexity of an alternative approach.

All in all, these considerations reduce the necessary steps to the prevention of race conditions. Since it has been decided to create multiple output files for parallel runs anyway, it is adequate to open a separate file per processor. A suitable procedure has been implemented using an adjusted file name format for the parallel case. This new format reads *tecout_YYYY_XXXX.dat* where *XXXX* still corresponds to the iteration but *YYYY* corresponds to the process which created the file. In fact, *YYYY* is always identical to the trailing part of the parallel input file and thus identifies the partiton.

Listing 5.6 shows a code fragment analogue to the complete implementation which can be found in appendix C. The main actions take place in lines 11 to 15 where a filename is assembled based on an output filename (stored in *fplt*) which contains the partition id. In line 16, the respective file is opened and is afterwards available for the exporting process.

A part of the possibilities offered by this macro can be found throughout this thesis as namely the figures 2.1, 2.3, 4.1, 4.2, 4.3 and 4.4 have been created with the aid of this macro.

```
1       character dump*17
2
3       {...}
4
5       if(pcomp(uct,'mac1',4)) then
6
7       {...}
8
9       elseif(pcomp(lct,'init',4)) then
10
11       write(dump,1999) nstep
12       if(pfeap_on) then
13        dump = fplt(LEN_TRIM(fplt)-4:LEN_TRIM(fplt)) // dump
14       endif
15       dump = 'tecout' // dump
16       open(unit=99, file=dump)
17
18       {...}
19
20      endif
21
22 1999 format(A,I3.3,'.',I3.3,A)
```

Listing 5.6: Modified access to Tecplot® output files in the parallel case

6 Performance in a massive parallel environment

Very large problems make high demands on the usable number of processors in order to keep the computing time a small as possible. Especially in applications where numerous results have to be available in a short time (e.g. optimisation) the number of processors available on a standard workstation (today, usually up to 4 processors) may not be sufficient. For these purposes, massive parallel environments like high performance computers or computer clusters offer an adequate performance provided that

1. The deployed software packages can be ported to the special circumstances which are often present in such environments

2. The available computing power can be used to full capacity due to an implementation which scales efficiently with the applied number of processors

This chapter approaches both mentioned issues from the point of view of a ParFEAP user. The first part is therefore dedicated to the necessary preparations and modifications for the application of ParFEAP in a massive parallel environment, namely the HHLR[1]. Afterwards, the performance and especially the scalability of the software package will be examined.

6.1 Special preparations for the use on the HHLR

The HHLR is a high performance computer manufactured by IBM running the UNIX based operating system IBM AIX 5.3[2]. Altogether, it currently consists of more than 500 processors which are grouped into 65 *nodes*. Each node is equivalent to a set of processors (usually eight) with access to a shared memory. This corresponds to the definition of a cluster of SMP systems as given in chapter 2.4.1. As a result, no network communication is required during a parallel run if eight or less processors are used (see figure 6.1). The HHLR allows the simultaneous request of up to 64 of these processors under usual conditions. These requests are processed by the so called LoadLeveler[3] which belongs to the IBM PE 4.3.1[4], a special implementation of the MPI for IBM AIX. The LoadLeveler prioritises incoming requests (also called *jobs*) according to predefined rules and executes a job when the necessary resources are idle. Consequently, interactive program executions are not possible on ordinary nodes which makes the use of scripts as introduced in chapter 5.1 inevitable. Installation and testing activities may of course take place on special *interactive nodes* where program execution is unrestricted (but usually slow, too).

But before the running of parallel jobs can be addressed in section 6.1.3, various particularities concerning the compilation of ParFEAP and the required packages on IBM AIX have to be resolved. In particular, most issues are related to the use of 64 Bit addressing, the special implementation of BLAS called ESSL[5], compilers incompatibilities and differing program names. The 64 Bit addressing is needed in order to be able to make full use of the available memory for large problems as a maximum of 4 GB can be addressed in standard 32 Bit programs. On the other hand, the use of ESSL instead of BLAS directly influences the performance of the computation as the basic algebraic instructions defined in BLAS have been optimised for the IBM AIX environment.

6.1.1 PETSc and related packages

The installation of PETSc with all required and optional components generally follows the same rules as outlined in chapter 3.1.2 and therefore mostly differing steps will be mentioned here. Of course, the mentioned particularities of the HHLR apply here too but fortunately PETSc includes predefined configuration settings for AIX systems. The accordant settings may simply be initialised via

```
export PETSC_ARCH=aix5.1.0.0-64
```

[1] Hessischer Hochleistungsrechner (Hessian High Performance Computer) – http://www.hrz.tu-darmstadt.de/hhlr/

[2] Advanced Interactive eXecutive – http://publib.boulder.ibm.com/infocenter/pseries/v5r3/index.jsp

[3] Tivoli Workload Scheduler LoadLeveler – http://www-03.ibm.com/systems/clusters/software/loadleveler/index.html

[4] IBM Parallel Environment – http://www-01.ibm.com/cgi-bin/common/ssi/ssialias?infotype=an&subtype=ca&htmlfid=897/ENUS206-250

[5] Engineering Scientific Subroutine Library – http://www-304.ibm.com/jct03004c/systems/p/software/essl/index.html

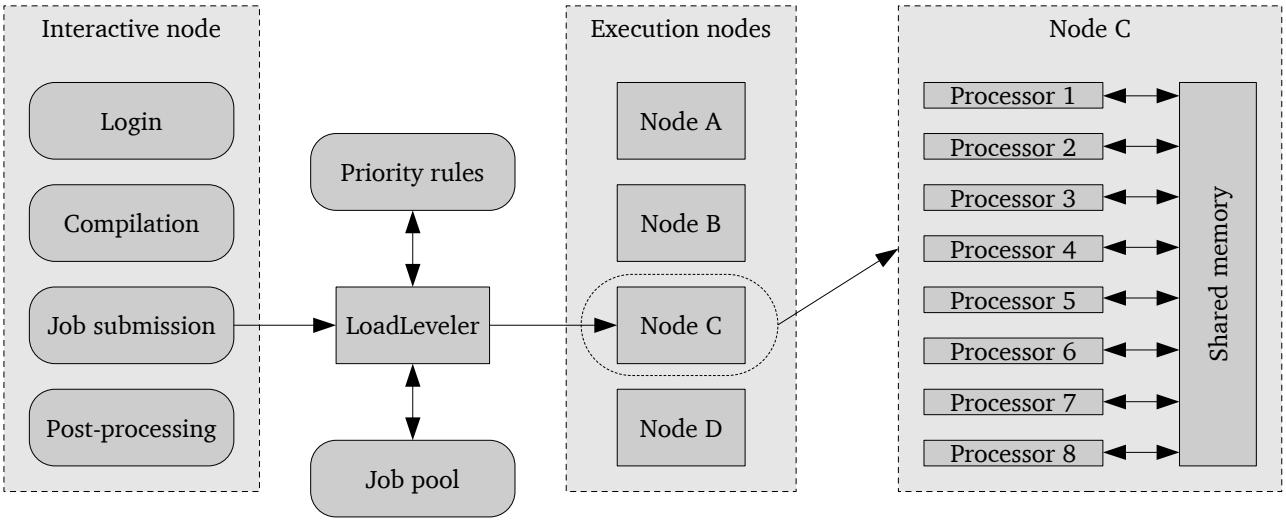


Figure 6.1: Simplified illustration of the layout of the HHLR

while some settings still have to be set manually. This especially refers to names and locations of various libraries and programs which generally depend on the encountered environment. But on the other hand, a part of the installation script of the current version is incompatible to the configuration of the HHLR, too. This is due to a non-standard syntax in the file *$PETSC_DIR/ python/ PETSc/ utilities/ debuggers.py*. Fortunately, version 2.3.2 of the PETSc distribution does not contain this flaw and the particular files is compatible. That way, the problem may temporarily be resolved by replacing the given file with the old version until an official bug fix is available.

As a next step, the configuration of PETSc has to be considered. In case of the HHLR, the following call (performed in *$PETSC_DIR*) led to the desired result:

```
./config/configure.py --with-cc='mpcc_r -q64' --with-fc='mpxlf_r -q64' --with-cxx='mpCC -q64' \
    --with-lapack-lib=/sw/lapack-3.1.1/lapack_AIX.a --with-ar='/usr/bin/ar -X64' \
    --with-blas-lib=/usr/lib/libessl.a --download-parmetis=1 --LDFLAGS=-b64 \
    --download-prometheus=1 --with-debugging=0 --with-batch=1 --with-mpi-shared=0
```

The given paths naturally are highly system dependent but nevertheless, this example should provide a good overview of required settings on IBM AIX systems while using 64 Bit addressing. Furthermore, the *–with-batch=1* option shows that batch environments make another change of the configuration process inevitable as they only allow the plain execution of certain programs (like compilers or *make*). Other executables like the PETSc test suite have to be started via MPI which is why the command named above will ask for a manual launch of the test suite at a certain point of the configuration. The accordant command reads

```
poe ./conftest -procs 1 -hostfile hostfile
```

where the command *poe* is the IBM version of the command *mpiexec* known from MPICH2, *-procs* defines the desired number of processes and *hostfile* is a text file which contains the word *localhost*. As soon as the test suite has been passed, the configuration process may be completed by means of the command:

```
./reconfigure.py
```

In some cases, the compilation of Prometheus takes too long and is aborted by PETSc (“Runaway process”). This error may be resolved by increasing the timeout defined in *$PETSC_DIR/python/PETSc/packages/Prometheus.py* next to the first occurrence of the word *timeout*.

The installation may then be finalised by

```
make all test
```

while it is needless to say that tests related to graphical features will fail due to the nonexistence of related packages on the HHLR.

6.1.2 FEAP and ParFEAP

Once the required packages have been installed successfully, the installation of FEAP itself can be engaged. Even though FEAP 8.2 in fact supports the encountered configuration, the task of installing the software package was rather complex and time-consuming since many of the steps presented in the following are not documented and had to be figured out by hand.

After *$FEAPHOME8_2* has been set up correctly, this task still requires several modifications of different files. First of all, the file *makefile.in* has to be adapted to the new environment. A summary of the changes in this file is displayed in listing 6.1. The majority of the applied changes corresponds to the switch to the 64 Bit mode and the required compilers

```
1 FINCLUDE = \$(FEAPHOME8\_2)/include -I\$(FEAPHOME8\_2)/include/integer8
2 FF = mpxlf_r -q64
3 CC = mpcc_r -q64
4 FFOPTFLAG = -O2
5 CCOPTFLAG = -O2
6 FOPTIONS = -malign-double -fno-second-underscore -funroll-loops -march=powerpc
7 AR = ar -rv -X64
8 MAKE = gmake
```

Listing 6.1: Required modifications of *makefile.in*

(lines 1 to 7). However, the last line addresses a special particularity and has been inserted by the author due to the fact that some of FEAP's makefiles use a syntax which is incompatible to the standard version of *make* available on IBM AIX. That is why the GNU implementation of *make* has to be used. The HHLR also supports this software package which may accessed via *gmake* which elucidates the purpose last line of the stated code. As almost all of FEAP's makefiles originally use the call *$(MAKE)* instead of the direct call *make*, this small change affects all accordant files. Unfortunately, the makefiles located in *$FEAPHOME8_2/elements* and *$FEAPHOME8_2/contact* call *make* directly and that is why each appearance of the command (but **not** each occurrence of the string) has to be replaced by *$(MAKE)* manually.

The files *$FEAPHOME8_2/unix/x11u.c* and *$FEAPHOME8_2/unix/cmem.c* configure the X windows driver and the dynamic memory allocation for the given system. For the use on an IBM operating system, the Fortran interface to some functions defined in this file have to be adjusted by removing the trailing underscore of each function definition near each appearance of the string *FORTRAN*.

Finally, FEAP provides a patch which resolves incompatibilities between the IBM and GNU Fortran compilers. This patch is located in *$FEAPHOME8_2/patch/ibm* and has to be incorporated into the main archive file *Feap8_2.a* which may be achieved with the aid of the *makefile* located in the parent directory. This file should be copied into the directory containing the patch while adapting the relative path to *makefile.in*. The subsequent execution of

```
gmake install
```

should eventually lead the desired result.

The executable files may then be created by carrying out the instructions given in chapter 3.2.2 while the outstanding changes read as follows:

- *gmake* has to be used instead of *make*.
- The expression *ar -rv* has to be replaced by *$(AR)* in *$FEAPHOME8_2/packages/arpack/archive/makefile* and *$FEAPHOME8_2/parfeap/packages/arpack/makefile*,
- In line 91 of *$FEAPHOME8_2/pndata.f* the comma succeeding the *write* command has to be removed

6.1.3 Job submission

Now that a working copy of ParFEAP has been obtained, it should be tested on an interactive node. For these purposes, the program may be launched from a directory containing a ParFEAP input file via

```
poe $FEAPHOME8_2/parfeap/feap -procs n -hostfile hostfile
```

where *n* is the number of processes and *hostfile* contains the word *localhost* at least *n* times (each occurrence in a new line).

If all tests have been passed, the preparations are completed and the LoadLeveler has to be used in order to take advantage of the full capacity of the architecture. To this end, each job has to be submitted to the LoadLeveler using the command

```
llsubmit jobfile
```

where *jobfile* is a special *command input file* defining several parameters of the request. An example of such an input file can be seen in listing 6.2. It includes most commonly required options but can not be exhaustive since these command files can become rather complex under special circumstances. A complete coverage of this topic can be found in [4].

```
1  #!/bin/ksh
2  #-----------------------------------------------------------
3  # @ output = run.log
4  # @ error = run.err
5  # @ notification = complete
6  # @ notify_user = johndoe@somewhere.org
7  # @ initialdir = /home/username/directory
8  # @ wall_clock_limit = 0:05:00
9  # @ job_type = parallel
10 # @ node = 2
11 # @ total_tasks = 16
12 # @ network.MPI = sn_all,,us
13 # @ resources = ConsumableCpus(1) ConsumableMemory(500MB)
14 # @ queue
15
16 $FEAPHOME8_2/parfeap/feap
```

Listing 6.2: Example LoadLeveler job submission file

The first line of the listing indicates that the input file in fact is a shell script with a special structure. As a result, it may make use of all features of shell scripts and especially those introduced in chapter 5.1. In this example, the only task of the script itself is to run ParFEAP using MPI. On the other hand, the LoadLeveler depends on additional information supplied via this file using a special syntax. Therefore, each line beginning with

```
# @
```

will be interpreted as an instruction. An explanation of the quoted instructions can be found in table 6.1. For the given example, this leads to the following major features:

- The run is a parallel run of ParFEAP using MPI with a total of sixteen processes on two nodes
- Each process consumes one processor and a maximum of 500 MB memory
- The computation will not take more than five minutes once the run has been started from */home/username/directory*
- When the job has been completed, an e-mail will be sent to *johndoe@somewhere.org* while the outputs can be found in *run.log* and *run.err* in the */home/username/directory* directory
- The run uses optimised network communication (“User Space” instead of “IP”; This setting is recommended for almost all cases but is only applicable if more than one node is used)

The settings provided in the command input file are crucial for the efficient execution of a job as the LoadLeveler has to ensure that all requested resources are available. Excessive values for *wall_clock_limit* or *ConsumableMemory* will thus usually cause unnecessary waiting times until the job can be started. In some cases, false settings will even prevent the job execution at all, especially when more processors are requested than available on the specified number of nodes. On the other hand, these values should not be chosen too small because the job would either be terminated before it has been completed or will fail because it cannot allocate enough memory. Moreover, the importance of the selection of the execution directory should not be underestimated. For example, program executions on the HHLR requiring many read and/or write operations should generally not be run from the home-directory but from a subdirectory of */global* where the access time is much shorter and a huge amount of disc space is available. This fact usually outweighs the missing of a backup of the contents of this directory.

Instruction	Purpose
output	Specifies a file to which all program output will be written
error	Same as *output* but for error messages
notification	Defines events triggering a notification for the user
notifiy_user	The e-mail address notifications should be sent to
initialdir	The directory from which the script will be executed
wall_clock_limit	Hard limit for the computing time. If a process has been running for the specified time period, it will be terminated by the LoadLeveler. Generally, smaller limits will lead to smaller queue times
job_type	Type of program execution (e.g. “serial” or “parallel”)
node	Exact number of nodes (groups of processors with shared memory access) or a range of acceptable nodes (*node = min,max*)
total_tasks	Number of processes that should be started
network.MPI	Controls the network communication between nodes concerning passed messages
resources	Defines the required memory and number of processors **per process**
queue	End of job input commands

Table 6.1: Summary of LoadLeveler instructions and their effects

6.2 Speed-up analysis

Besides the correctness of results obtained during parallel runs (see chapter 4), the actual reduction of the computing time is the most important aspect of ParFEAP. Therefore, several facets of this topic have been analysed using different criteria and solver configurations. To this end, the problem presented in chapter 4.3 served as a benchmark test (The respective input file can found in appendix A.2). It has been chosen in order to show the general trends concerning the differences between standard and multigrid solvers as well as between different mesh sizes. On this account, the presented results refer to the plain solution time without partitioning (that is, the time period from the beginning of the computation until the last processors has finished) if not otherwise stated. The numerical settings correspond in all cases to those already given in table 4.1.

6.2.1 Serial compared to two-processor performance

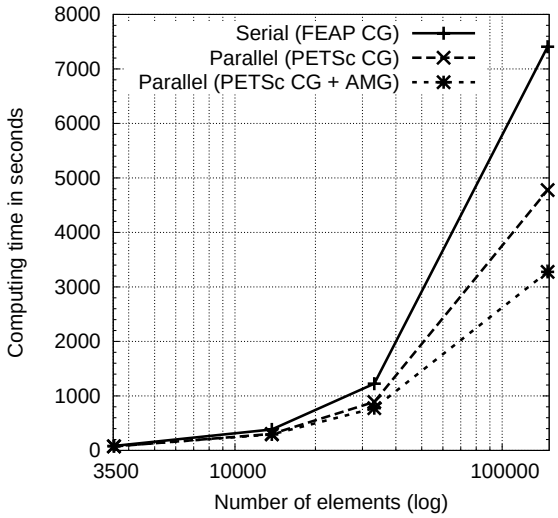
From the point of view of the user, the main focus is on the question whether the extra lead time for using ParFEAP instead of FEAP is justified. This section tries to answer this question for a standard workstation configuration where nowadays two processors are often present. Consequently, the computing times and the respective performance indicators (see 2.4.3, in this case with $N = 1$) for a sequences of (with respect to the number of elements) increasing mesh sizes have been determined on one and two processors.

The acquired results are visualised in figure 6.2(a) which includes the parallel results for the standard solver (Conjugate gradients, CG) and an algebraic multigrid solver (AMG). The plot indicates that the use of ParFEAP on two processors is not reasonable for small problems as the absolute gain is negligible due to a high proportion of ghost cells and the emerging communication effort (see figure 6.2(b)). However, the gain grows rapidly with an increasing mesh size which is exemplified by a speed-up of 1.55 on the finest mesh using the CG solver. This result is even excelled by the AMG solver with a speed-up of 2.26 which means that the execution time has been more than halved which of course is only possible due to the general performance benefits of AMG solvers.

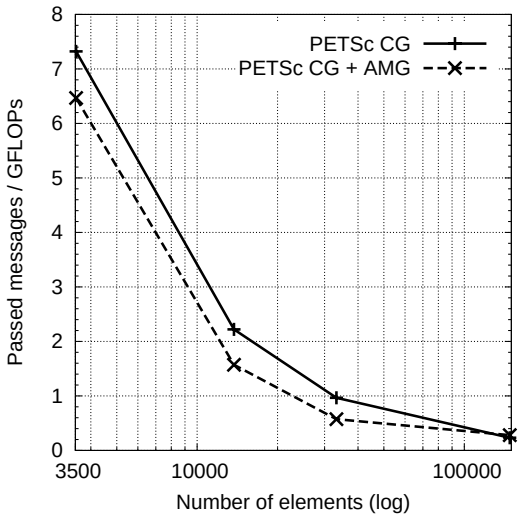
In terms of practical applications (where the number of elements is usually at least as big as in the finest featured mesh), this result strongly encourages the adoption of ParFEAP. Additionally, the power of Prometheus has become evident while it still has to be verified whether similar findings apply to the use of more processors as smaller partitions generally increase the number of passed messages disproportionately for multigrid solvers (see [8]).

6.2.2 Scaling with an increasing number of processors

Another important characteristic of a parallel software tool is the trend of the computing times when the number of processors rises. Especially in massive parallel environments like the HHLR where one tries to use as many processors as possible in order to minimise standby times. On the other hand, such environments typically enforce some kind of load leveler which prioritises jobs using few processors. Consequently, more processors lead to a longer waiting period which is only acceptable if this can be compensated by a sufficient acceleration of computation. In this context, the parallel



(a) Computing time



(b) Ratio of passed messages to arithmetic operations during a parallel run

Figure 6.2: Plots concerning a progression of increasing element counts for a problem as introduced in chapter 4.3 on one and two processors

efficiency plays an important role as low values indicate that yet smaller partitions will not decrease the computing time anymore due to the dominance of the additional communication effort.

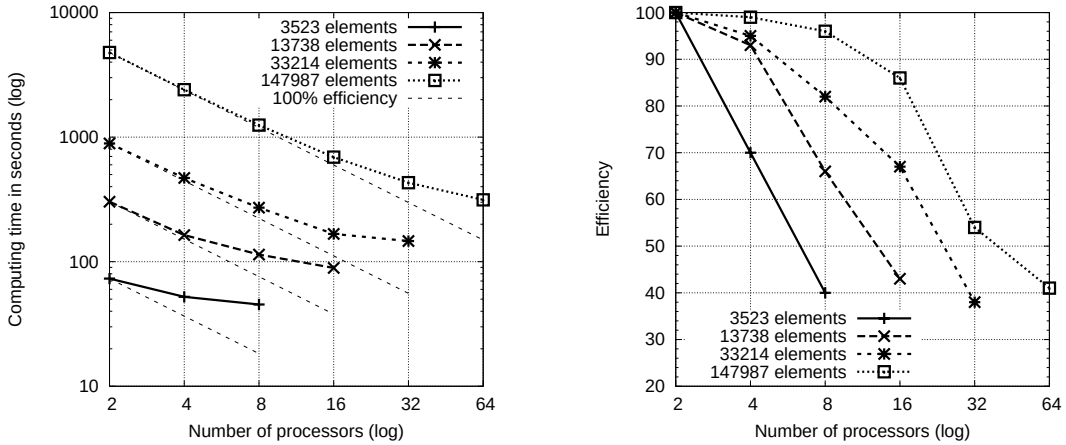
The characteristic scaling of ParFEAP was therefore analysed by measuring the computing time for a each time doubling number of partitions (and processors) while keeping all other numerical settings constant. This means that the parallel run on two processors was used as a reference run which rules out the influence of the different solvers used in FEAP and ParFEAP (accordingly, the definitions given in chapter 2.4.3 have been used for $N = 2$). The obtained results can be taken from figure 6.3 for the standard CG solver and figure 6.4 for the CG solver with a multigrid preconditioner. Here, it becomes apparent that the reasonable number of processors increases with the problem size but that the asymptotic efficiency approaches zero in all cases. The course of this development is obviously different for the standard and the multigrid case. In contrast to the fact that the absolute computing times are much smaller using few processors, the multigrid advantage gradually disappears since the parallel efficiencies decrease considerably faster for this configuration (figure 6.4(b)). Another evidence for this finding is the strong deviation of the measured computing times from the ideal case in figure 6.4(a).

All in all, ParFEAP showed a quite decent performance throughout the scaling tests which may be underlined by a numerical example: Compared to the serial case, the execution on the finest mesh could be speed-up by a factor of approximately 11 for both solver configurations when using 16 processors. In other words, this means that the mentioned computation took eleven minutes instead of more than two hours which should generally justify the use ParFEAP in applicable situations. The use of Prometheus did not cause any problems but did not improve the results for massive parallel runs either. Nevertheless, a remarkable acceleration of up to 31% compared to the standard CG method could be achieved on the finest mesh for small numbers of processors.

6.2.3 Influence of partitioning

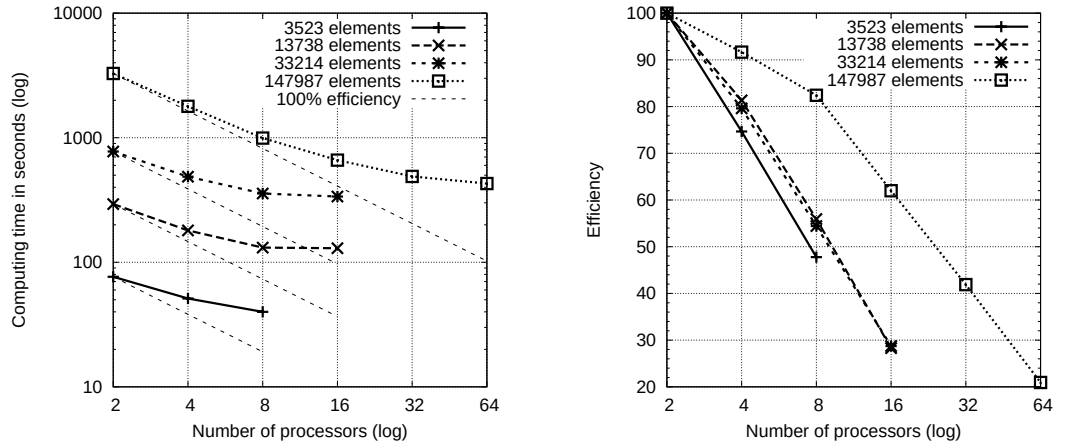
Only plain solution times have been considered up to this point and thus the effect of the partitioning has been neglected. In the context of the parallel runs, this means that the total processing time has been underestimated. We will therefore show in the following that the influence of the partitioning is small and, even more important, does not deviate the conclusions in terms of scaling characteristics. This might be the case if the partitioning times are long compared to the computing times **and** are highly dependent on the number of partitions at the same time. Another aspect in terms of partitioning times is the acceleration when using ParMETIS instead of METIS which will also be discussed in this section. Finally, an attempt to separate the load balancing efficiency from the efficiencies (see chapter 2.4.3) displayed in figures 6.3(b) and 6.4(b) will conclude this chapter.

The serial partitioning times are presented by figure 6.5(a) where a considerable increase of the time duration becomes noticeable which is especially true for the finest mesh. However, the maximum absolute deviance is equal to six seconds



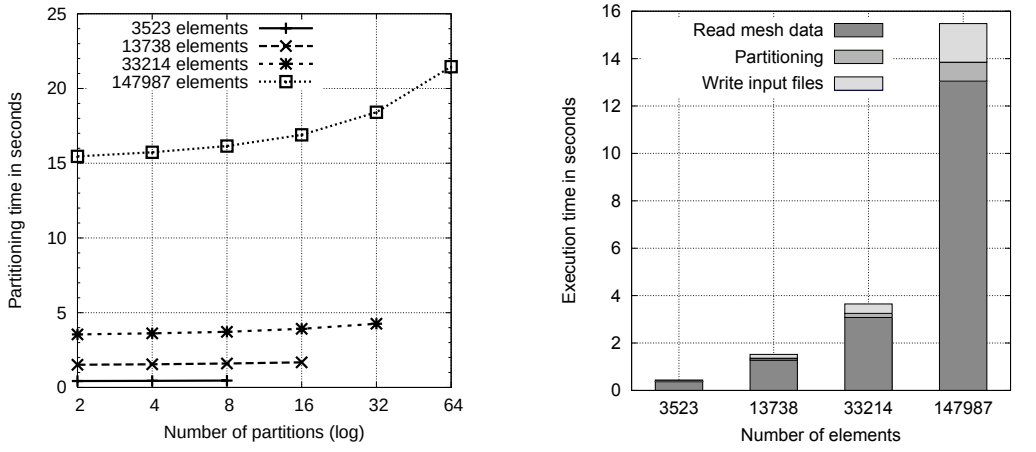
(a) Comparison of computing times (b) Progression of overall efficiencies

Figure 6.3: Scaling test results using a standard CG solver



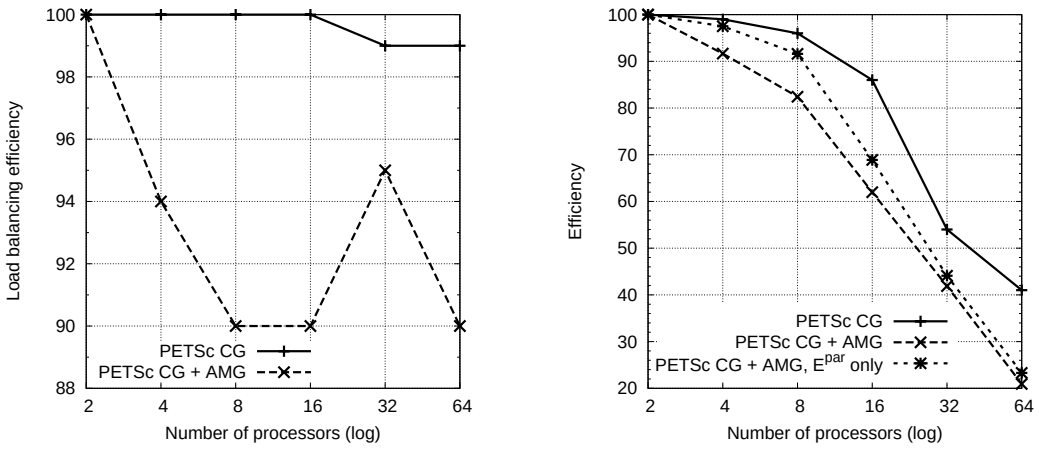
(a) Comparison of computing times (b) Progression of overall efficiencies

Figure 6.4: Scaling test results using a CG solver in combination with a multigrid preconditioner



(a) Progression of overall times (b) Breakdown by subtask (two processors)

Figure 6.5: Timing results for the mesh partitioning of the given test case using a serial graph partitioner (METIS)



(a) Comparison of solvers (b) Influence on the overall efficiency

Figure 6.6: Analysis of the load balancing efficiency on the finest mesh

which accounts for less than 1% of the total computing time in this case. This leads to the conclusion that the disregard of the influence of the partitioning in chapter 6.2.2 was legitimate.

Regrettably, no comparable data could be collected for the use of ParMETIS. This is due to the fact that no approach could be found which allows to cope with the variable number of processors required by the ParMETIS shortcut (see appendix B.2) in the batch environment on the HHLR. On the other hand, controlling the parallel partitioning manually requires a total of two serial runs which is rather inefficient which fact is underlined by figure 6.5(b). It shows the proportions of the subtasks that form a serial partition run and one can see that the reading of the mesh data is by far the most time consuming part making out about 84% of the total time in all four cases. This has two consequences:

1. When calling the parallel partitioner manually, the mesh has to be read twice which doubles the mesh input time. As less than 3% of the program execution remains parallelisable under such circumstances, this approach will always be slower than the use of METIS.

2. In a similar configuration, ParMETIS cannot offer a significant advantage compared to METIS.

Considerations concerning the quality of subdivisions generated by both methods can be carried out simultaneously given that both software packages base on the same algorithm (see [6] and [7]). Even though the resulting partitions are not always identical, no differences in terms of the load balancing efficiency could be ascertained. The load balancing efficiency is a natural benchmark for the quality of the spatial decomposition (besides a preferably small number of ghost cells) since differences in the computing times on different processors are directly related to an unbalanced distribution of elements. Nevertheless, one has to keep in mind that an identical number of elements per partition does not guarantee an optimal behaviour because computing times also depend on other characteristics, in particular when using a different order of the ansatz functions or another solver type.

The load balancing efficiencies have been determined for all mesh sizes and both solver configurations. Throughout all tests for the standard CG solver, the results were virtually ideal while, in contrast, the use of Prometheus lead to significant deviations from the optimum (see figure 6.6(a)). This partly explains the discrepancy in terms of efficiency between the standard and the multigrid solver which has been stated earlier. The influence of the this fact can be seen in figure 6.6(b) where the actual efficiencies have been opposed to the theoretical case of no distortion through an imperfect balance of workload. Here, one can see that an optimised partitioning algorithm could considerably improve the multigrid performance for small and medium numbers of processors but that the general tendency remains unaffected.

7 Conclusion

During the course of this thesis, the parallelised version of the FEM toolkit FEAP has been examined with respect to the:

- required installation steps
- validity of obtained results and emerging limits of application
- ease of use and possible improvements
- parallel performance

The following sections evaluate the basic findings on these topics which should help a potential user to judge whether ParFEAP is suitable for his needs and whether it supports all required features. Finally an outlook on the possible future trend will be given.

7.1 Evaluation of FEAP 8.2

The overall impression of the software package is quite positive despite some flaws which especially exist in the installation procedure and the sparse documentation of some topics. But once a viable installation of ParFEAP has been obtained its utilisation is straightforward, reliable and generally speeds up computations considerably. Unfortunately, this can only guaranteed for standard features as it has been shown that at least contact formulations with rigid bodies and finite deformations for plastic material models are not supported in the current version. This fact may be considered as the major disadvantage of ParFEAP as it totally prevents its use for accordant problems.

On the other hand, many other aspects exist which encourage the use of ParFEAP. First of all, serial input files may easily be adopted for the parallel case (and vice versa) and thus a smooth switch-over is possible. In combination with the observed gain of performance this makes ParFEAP an attractive option. Especially in fields of numerical research, the accessibility of PETSc introduces another benefit since other solvers may be integrated relatively easily using the sophisticated framework offered by this software package. Lastly, it shall be mentioned that ParFEAP's drawbacks concerning the simple creation of professional plots and the use in batch environments could be resolved (see chapter 5) which further increases its effectiveness.

7.2 Prospect

As already indicated in the introduction, the author believes that the efficient parallelisation of computation is a necessary task considering the growing demand for fast and accurate analyses of big problems by means of the FEM. This applies in particular to the propagation of optimisation techniques in practical applications which strongly relies on the efficient solving of many variations of a single problem in a short time. The linking of ParFEAP to accordant algorithms might thus be an interesting subject for further investigations.

Another possible field of application is the efficient modeling of micropolar and micromorph continua which is a field of research at the TU Darmstadt [1]. This approach aims at reproducing certain *length scale effects* which occur in metals under plastic deformation using special element formulations. The applicability of these formulations is currently strongly limited by the high computational effort which could be evaded by the use of ParFEAP instead of FEAP. However, the lack of contact capabilities in the current version still renders some uses impossible and it has to be seen whether future versions will add this feature. Anyway, this field of research is just an example of a number of ongoing topics where the practical application is restricted by performance issues which could be resolved by the use of parallelisation.

[1] http://www.fnb.tu-darmstadt.de/de/forschung/projekte/bauer/bauer.php

Bibliography

[1] S. Balay, K. Buschelmann, V. Eijkhout, et al. *PETSc Users Manual*. Argonne National Laboratory Mathemathics and Computer Science Division, Argonne, May 2007.

[2] W. Gropp, E. Lusik, D. Ashton, et al. *MPICH2 User's Guide Version 1.0.7*. Argonne National Laboratory Mathemathics and Computer Science Division, Argonne, April 2008.

[3] IBM Corporation. *XL Fortran Enterprise Edition for AIX User's Guide Version 9.1*, 3 edition, August 2004.

[4] IBM Corporation. *Tivoli Workload Scheduler LoadLeveler Using and Administering Version 3 Release 4*, October 2006.

[5] IBM Corporation. *IBM Parallel Environment for AIX and Linux Installation Version 4 Release 3.1*, June 2007.

[6] G. Karypis and V. Kumar. *METIS A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices Version 4.0*. University of Minnesota Department of Computer Science and Engineering, Minneapolis, September 1998.

[7] G. Karypis, K. Schloegel, and V. Kumar. *ParMETIS Parallel Graph Partitioning and Sparse Matrix Ordering Library Version 3.1*. University of Minnesota Department of Computer Science and Engineering, Minneapolis, August 2003.

[8] B. Metsch. *Ein paralleles graphenbasiertes algebraisches Mehrgitterverfahren*. PhD thesis, Rheinische Friedrich-Wilhelms-Universität Bonn, September 2004.

[9] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), 1965.

[10] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes The Art of Scientific Computing*. Cambridge University Press, Cambridge, 2007.

[11] M. Schäfer. *Computational Engineering Introduction to Numerical Methods*. Springer Verlag, Berlin, 2006.

[12] K. Stüben. Algebraic multigrid (amg) An introduction with applications. In *GMG Report No. 53*. GMD – Forschungszentrum Informationstechnik GmbH, March 1999.

[13] H. Sutter. The concurrency revolution. *C/C++ Users Journal*, 23(2), 2005.

[14] R. L. Taylor. *FEAP – A Finite Element Analysis Program Version 8.2 Example Manual*. University of California at Berkeley Department of Civil and Environmental Engineering, Berkeley, March 2008.

[15] R. L. Taylor. *FEAP – A Finite Element Analysis Program Version 8.2 Installation Manual*. University of California at Berkeley Department of Civil and Environmental Engineering, Berkeley, March 2008.

[16] R. L. Taylor. *FEAP – A Finite Element Analysis Program Version 8.2 Programmer Manual*. University of California at Berkeley Department of Civil and Environmental Engineering, Berkeley, March 2008.

[17] R. L. Taylor. *FEAP – A Finite Element Analysis Program Version 8.2 Theory Manual*. University of California at Berkeley Department of Civil and Environmental Engineering, Berkeley, January 2008.

[18] R. L. Taylor. *FEAP – A Finite Element Analysis Program Version 8.2 User Manual*. University of California at Berkeley Department of Civil and Environmental Engineering, Berkeley, March 2008.

[19] R. L. Taylor and S. Govindjee. *FEAP – A Finite Element Analysis Program Version 8.2 Parallel User Manual*. University of California at Berkeley Department of Civil and Environmental Engineering, Berkeley, March 2008.

[20] Tecplot, Inc., Bellevue, Washington. *Tecplot® User's Manual Version 10*, March 2005.

A FEAP input files for parallel validation

All validation tests where conducted using the problem definitions stated below. They are based on problems provided by the supervisor and have been adjusted in order to examine different features of FEAP. Additionally, the files shown in section A.2 have been used for the timing tests in chapter 6, while the solution commands have been changed slightly in order to minimise the influence of output operations. The corresponding serial input files are not displayed as they are identical to the parallel input files if merged into a single file (see chapter 3.3.2).

A.1 Pierced plate under static load using linear elastic material model

```
1  FEAP
2    0 0 0 3 3 4
3
4  INCLude mesh              ! Import (unstructured) mesh data (Ansys mesh export)
5
6  EBOUndary
7    2 200 1 1 1             ! Fixate upper boundary
8
9  EFORce
10   2 0 0 -50.0 0
11
12 PARAmeters
13   mu = 81000              ! Shear modulus
14   k = 160000              ! bulk modulus
15   nu = (3*k - 2*mu)/(2*mu + 6*k)
16                           ! poisson's ratio
17   em = 2*(1+nu)*mu        ! youngs modulus
18
19 MATErial, 1
20   SOLId
21   ELAStic ISOTropic em nu
22
23 END
24
25 BATCh
26   GRAPh,,2
27   OUTDomains
28 END
29
30 STOP
```

Listing A.1: File *itest1parallel*

```
1  BATCh
2    FORM
3    TANG,,1
4    SOLVe
5    STRE,NODE
6    DISP,NODE
7  END
```

Listing A.2: File *solve.test1parallel*

A.2 Pierced plate under transient load with plastic material model

```
FEAP
  0 0 0 3 3 4

INCLude mesh              ! Import (unstructured) mesh data (Ansys mesh export)

PARAmeters
  dy = 4                  ! Parameter for the prescribed displacement

EBOUndary
  2 200 1 1 1
  2 0   0 1 0

EDISplacement
  2 0 0 -dy 0             ! Prescribe y discplacement to the lower boundary

EPROportional
  2 0 0 1 0               ! Apply the displacement step by step
                          ! See solve.test2parallel for exact characteristics

PARAmeters
  mu = 81000              ! Shear modulus
  k = 160000              ! Bulk modulus
  nu = (3*k - 2*mu)/(2*mu + 6*k)
                          ! Poisson's ratio
  em = 2*(1+nu)*mu        ! Young's modulus
  y = 450.                ! Uniaxial yield stress

MATErial, 1
  SOLId
  ELAStic ISOTropic em nu
  PLAStic MISEs y         ! Enable plasticity

END

BATCh
  GRAPh,,2
  OUTDomains
END

STOP
```

Listing A.3: File *itest2parallel*

```
BATCh
  PROP,,1                 ! Begin the definition of the transient load
END
  2, 2                    ! Type 2 definiton, 2 entries ("point in time" "portion")
  0 0 1 1                 ! Linear rise from no to full displacment (defined by dy)

BATCh
  DT,,0.01                ! Define size of timesteps in seconds
  LOOP,time,50            ! 50 time steps
    TIME
    LOOP,augment,5
      LOOP,newton,30
        TANG,,1
```

```
14          NEXT
15          AUGMent
16        NEXT
17        STRE,NODE
18        DISP,NODE
19      NEXT
20  END
```

Listing A.4: File *solve.test2parallel*

A.3 Contact of linear elastic block with rigid cylinder

```
1   FEAP
2     0 0 0 2 2 4
3
4   PARAmeters
5     a = 40                ! Block length
6     b = 10                ! Block height
7     c = 32                ! Block divisions in x direction
8     d = 8                 ! Block divisions in y direction
9     r = 8                 ! Radius of penetrating cylinder
10    di = 2                ! Maximum displacement
11    c1 = c+1
12    d1 = d+1
13
14  BLOCk                   ! Block-shaped structured grid
15    CARTesian c d 1 1 1
16    1 -a/2 -b-r
17    2 a/2 -b-r
18    3 a/2 0-r
19    4 -a/2 0-r
20
21  EBOUndary
22    2 -b-r 0 1            ! Fix lower boundary in y direction
23
24  EDISplacement
25    2 -b-r 0.0 di
26
27  CBOUnd,ADD              ! Add boundary conditions instead of overwriting
28    NODE 0.0 -b-r 1 0     ! Fix middle nodes in x-direction
29    NODE 0.0 -r 1 0
30
31  PARAmeters
32    mu = 81000            ! Shear modulus
33    k = 160000            ! Bulk modulus
34    nu = (3*k - 2*mu)/(2*mu + 6*k)
35                          ! Poisson's ratio
36    em = 2*(1+nu)*mu      ! Young's modulus
37    y = 450.              ! Uniaxial yield stress
38
39  MATErial, 1
40    SOLId
41    ELAStic ISOTropic em nu
42    PLAStic MISEs y       ! Enable plasticity
43    MIXEd                 ! mixed element formulation
44
45  END
```

```
46
47 CONTact,ON
48
49 SURFace 1                ! Definition of the first contact surface
50   LINE 2
51   FACEts
52   1 -1 d1*c1 (d1*c1)-1
53   c 0  (d1*c1)-c+1 (d1*c1)-c
54
55 SURFace 2
56   RIGId                   ! Second surface is a rigid cylinder
57   FUNCtion CYLInder 1,r,0,0
58
59 PAIR 1                   ! Define contact conditions
60   NTOR 1 2
61   SOLM PENAlty 2.e+3 2.e+3
62   AUGMent
63   TOLE,,1.e-5 1.e-5 1.e-5
64
65 END
66
67 BATCh
68   GRAPh,,2
69   OUTDomains
70 END
71
72 STOP
```

Listing A.5: File *itest3parallel*

```
1  BATCh
2    PROP
3    DT,,0.1
4  END
5    2 3
6    0 0 0.8 1 1 0
7
8  BATCh
9    LOOP,time,10
10     TIME
11     LOOP,augment,5
12       LOOP,newton,30
13         TANG,,1
14       NEXT
15       AUGMent
16     NEXT
17     STRE,NODE
18     DISP,NODE
19   NEXT
20 END
```

Listing A.6: File *solve.test3parallel*

B ParMETIS shortcut

As indicated in chapter 3.3.2, ParFEAP offers a convenient way of calling ParMETIS from ParFEAP if the standalone partitioner has been compiled. Listing B.1 shows the syntax of this call which should make ParFEAP create a plain version of the input file, run ParMETIS on the specified number of processors and finally create the input files for the solution run (see figure B.1).

```
1 BATCh
2   OUTMesh
3   GRAPh PARTition 2
4   OUTDomains
5 END
```

Listing B.1: Syntax for a direct call of ParMETIS from ParFEAP in a partition run using two partitions

Unfortunately, the implementation of the given shortcut contains a bug which makes small modifications of the source code necessary. An extract of the corresponding source code is given in listing B.2. The problematical parts are:

- The current of version of MPICH2 creates an executable named *mpiexec* but in line 234 FEAP tries to call *mpirun*. Additionally, the command does not include the full path the executable and thus assumes global access to the executable.
- In line 239, the identifier *$FEAPHOME8_1* is not valid anymore for the current version.
- The call of *partition* is incomplete as the filename of the output file is not specified.

```
219           if(pcomp(lct,'part',4)) then
220
221 c           Copy input file to Gfile
222
223             i               =  index(finp,' ')
224             gfile(1:3)      = 'cp '
225             gfile(4:i+4)    = finp(1:i)
226             gfile(i+3:i+6)  = '.rev'
227             gfile(i+7:i+12) = ' Gfile'
228             call system (gfile)
229
230 c           Set file command to run partitioning using ParMETIS
231
232             gfile( 1:15) = 'mpirun -np   2 '
233             write(gfile(12:14),'(i3)') nint(ct(1)) ! Number of processors
234
235 c           Location of file 'partition using FEAPHOME8_1 definition'
236
237             gfile(16:79) = '$FEAPHOME8_1/parfeap/partition/partition '
238             gfile(80:89) = '   2 Gfile'
239             write(gfile(81:83),'(i3)') nint(ct(1)) ! Number of domains
240
241 c           Execute 'partition'
242
243             call system (gfile)
```

Listing B.2: Extract from *pmacr7.F* located in *$FEAPHOME8_2/parfeap*

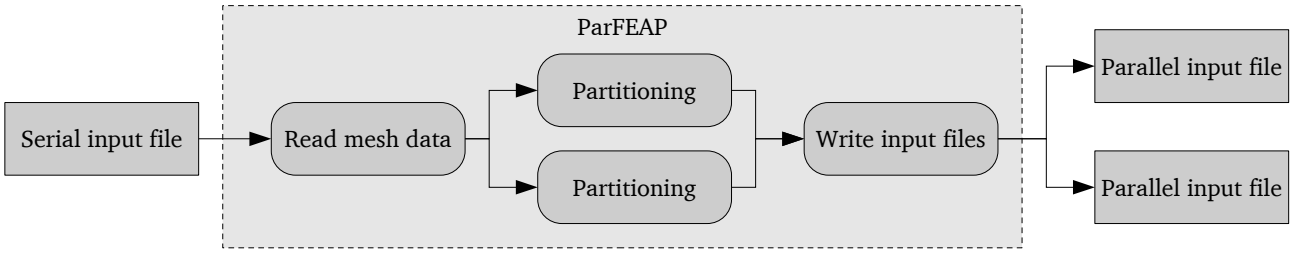


Figure B.1: Workflow of a partition run with a syntax as given in listing B.1

B.1 Bug-fix for the use in an interactive environment

The easiest way of coping with the first issue is the creation of an alias for the *mpiexec* command named *mpirun*. Still assuming a BASH (see chapter 3), this can for example be achieved via

```
alias mpirun='$PETSC_DIR/externalpackages/mpich2-1.0.5p4/bin/mpiexec'
```

while the absolute path has to be adjusted to the present system. Of course, this is only possible if the alias *mpirun* is not already in use. In this case the corresponding part of line 234 could be replaced by the desired name while it has to be noted that changes of the string length make modifications of the lines 235 to 241 necessary.

A fix for both other bugs can be found in listing B.3. In line 243, the false identified has been replaced and the missing filename has been added (cf. listing B.2). As a result, the string *gfile* has become longer which is why the declaration in line 64 has been updated (see listing B.4, the length has been changed from 89 to 101).

```
238 c             Location of file 'partition using FEAPHOME8_2 definition'
239
240               gfile(16:79) = '$FEAPHOME8_2/parfeap/partition/partition '
241               gfile(80:89) = '   2 Gfile'
242               write(gfile(81:83),'(i3)') nint(ct(1)) ! Number of domains
243               gfile(90:101) = ' graph.file'
```

Listing B.3: Extract of a modified version of listing B.2

```
64        character  gfile*101,gplf*128,fext*5,c*1
```

Listing B.4: Modified string length declaration for listing B.3

When all modifications have been completed, ParFEAP may be recompiled and finally the syntax shown in listing B.1 should be applicable.

B.2 Difficulties arising from an execution using the LoadLeveler

In situations where an interactive program execution is not possible, the particularities of ParFEAP evoke serious problems. In order to start a partition run, ParFEAP has to be started using one process (see figure B.1) but when using the LoadLeveler, this is only possible if one processor is requested. Unfortunately, this prohibits ParFEAP from running *partition* on multiple processors because only one processor will be assigned.

During the course of thesis, no simple workaround for this contradiction (e.g. using a shell script) could be found. Nevertheless, it should generally be possible to modify FEAP's source files in order to create an auxiliary executable which always runs in partition mode and thus may be started on more than one processor. That way, one could also implement a parallel version of the *GRAPh* command at the same time which would make the parallel graph partitioner more valuable.

C Modified FEAP user macro

This chapter consists of a listing of a modified version of a FEAP macro originally written by the supervisor. In order to create a built of FEAP using this macro, it is necessary to place the code stated below in *$FEAPHOME8_2/ user/ umacr1.f*. Lines containing changes by the author can be identified by a trailing star.

```
1  subroutine umacr1(lct,ctl,prt)
2
3  c       * * F E A P * * A Finite Element Analysis Program
4
5  c....   Copyright (c) 1984-2007: Regents of the University of California
6  c                                 All rights reserved
7
8  c-----[--.----+----.----+----.-----------------------------------------]
9  c      Modification log                                    Date (dd/mm/year)
10 c        Original version                                       01/11/2006
11 c                                                    bauer      08/01/2008
12 c        added support for various tets/hexes        bauer      13/06/2008
13 c        added support for parallel runs             mueller    15/07/2008      *
14 c-----[--.----+----.----+----.-----------------------------------------]
15 c       Purpose: Data export for convenient postprocessing in TECPLOT.
16 c
17 c                Tested for:
18 c
19 c                4-Node quadrilaterals
20 c                8/20/27-node hexahedrals
21 c                4/10/11-node tetrahedrals
22
23
24 c       Inputs:
25 c          lct       - Command character parameters (init,write,close)
26 c          ctl(3)    - Command numerical parameters (none)
27 c          prt       - Flag, output if true (none)
28
29 c       Outputs:
30 c          tecout,cpuidx,nstep,.dat.                                        *
31 c                    - tecplot readable datafile(s) containing:
32 c                                - X, Y, (Z)
33 c                                - DX, DY, (DZ)
34 c                                - UX, UY, (UZ)
35 c                                - FX, FY, (FZ)
36 c                                - 1-PR, 2-PR, 3-PR
37 c                                - I-1, J-2, J-3
38
39 c       Example for macro usage:
40
41 c        BATCh
42 c        LOOP,time,10
43 c          TECP,init             ! open output unit
44 c          TIME
45 c          LOOP,newton,30
46 c            TANG,,1
47 c          NEXT
48 c          STRE,NODE             ! compute nodal stresses (mandatory!)
```

```
c          TECP, write          ! write postprocessing data
c          TECP, close          ! close ouput unit
c        NEXT

c-----[--.----+----.----+----.-----------------------------------------]
      implicit  none

      include  'iofile.h'  ! standard i/o channels
      include  'ioincl.h'  ! additional i/o declarations
      include  'counts.h'  ! important counter variables
      include  'umac1.h'   ! macro names
      include  'pointer.h' ! pointers in blank common
      include  'comblk.h'  ! blank common
      include  'cdata.h'
      include  'sdata.h'
      include  'comfil.h'  ! input files names                              *
      include  'pfeapb.h'  ! information on parallel solutions              *

      logical   pcomp, prt
      integer   j
      character lct*15,dump*17                                              *
      real*8    ctl(3)

      save
c     Set command word

      if(pcomp(uct,'mac1',4)) then       ! Usual form
       uct = 'tecp'                      ! Specify 'name'
      elseif(urest.eq.1) then            ! Read restart data

      elseif(urest.eq.2) then            ! Write restart data

      elseif(pcomp(lct,'init',4)) then   ! open output unit; incl. counter

c     Try to open file for later use (filename based on input file)         *
       write(dump,1999) nstep                                               *
c     Identify partition in the parallel case                               *
       if(pfeap_on) then                                                    *
        dump = fplt(LEN_TRIM(fplt)-4:LEN_TRIM(fplt)) // dump                *
       endif                                                                *
       dump = 'tecout' // dump                                              *
       open(unit=99, file=dump)                                             *

      elseif(pcomp(lct,'close',4)) then  ! close output unit
       close(unit=99)

      elseif(pcomp(lct,'write',4)) then ! output postprocessing data

c     Write nodal postprocessing data
c     2D Data
       if(ndm.eq.2) then
         write(99,*) 'TITLE = \"FEM DATA, FEAP-->TECPLOT\"'
         write(99,*) 'VARIABLES=\"X\",\"Y\",\"DX\",\"DY\",\"UX\",\"UY\",
     &\"FX\",\"FY\",\"1-PR\",\"2-PR\",\"3-PR\",\"I-1\", \"J-2\",
     &\"J-3\"'

         write(99,2001) numnp,numel
         if(nen.eq.4) then
```

```
107             write(99,*) 'DATAPACKING=POINT, ZONETYPE=FEQUADRILATERAL'
108           elseif(nen.eq.3) then
109             write(99,*) 'DATAPACKING=POINT, ZONETYPE=FETRIANGLE'
110           endif
111
112           do j = 0,numnp-1
113                 write(99,2002)
114       &       hr(np(43)+j*ndm),hr(np(43)+j*ndm+1),  ! nodal coor X 43
115       &       hr(np(40)+j*ndf),hr(np(40)+j*ndf+1),  ! nodal disp U 40
116       &       hr(np(43)+j*ndm)+hr(np(40)+j*ndf),    ! deformed nodal pos.
117       &       hr(np(43)+j*ndm+1)+hr(np(40)+j*ndf+1),
118       &       hr(np(27)+j*ndm),hr(np(27)+j*ndm+1),  ! forces,disp F 27
119       &       hr(np(57)+numnp*1+j),                 ! principal stress 1
120       &       hr(np(57)+numnp*2+j),                 ! principal stress 2
121       &       hr(np(57)+numnp*3+j),                 ! principal stress 3
122       &       hr(np(57)+numnp*4+j),                 ! I 1
123       &       hr(np(57)+numnp*5+j),                 ! J 2
124       &       hr(np(57)+numnp*6+j)                  ! J 3 (v. Mises-Stress)
125           end do
126 c         'Array 33 IX: Element Connections'
127           if(nen.eq.4) then     !quadrilateral
128             do j = 0,numel-1
129                   write(99,2003) mr(np(33)+j*nen1),mr(np(33)+j*nen1+1),
130       &                          mr(np(33)+j*nen1+2),mr(np(33)+j*nen1+3)
131             end do
132           elseif(nen.eq.3) then !triangle
133             do j = 0,numel-1
134                   write(99,2004) mr(np(33)+j*nen1),mr(np(33)+j*nen1+1),
135       &                          mr(np(33)+j*nen1+2)
136             end do
137           end if !nen
138
139 c      3D Data
140         elseif((ndm.eq.3).and.(ndf.eq.3)) then !classic continuum elem.
141
142           write(99,*) 'TITLE = \"FEM DATA, FEAP->TECPLOT\"'
143           write(99,*) 'VARIABLES=\"X\",\"Y\",\"Z\",\"DX\",\"DY\",\"DZ\",
144       &\"UX\",\"UY\",\"UZ\",\"FX\",\"FY\",\"FZ\",\"1-PR\",\"2-PR\",\"3-PR
145       &\",\"I-1\", \"J-2\",\"J-3\"'
146           write(99,2001) numnp,numel
147           if(nen.eq.8 .or. nen.eq.27 .or. nen.eq.20) then
148             write(99,*) 'DATAPACKING=POINT, ZONETYPE=FEBRICK'
149           elseif(nen.eq.4 .or. nen.eq.10 .or. nen.eq.11) then
150             write(99,*) 'DATAPACKING=POINT, ZONETYPE=FETETRAHEDRON'
151           end if !nen
152
153           do j = 0,numnp-1
154               write(99,2012)
155       &       hr(np(43)+j*ndm),hr(np(43)+j*ndm+1),hr(np(43)+j*ndm+2),
156       &       hr(np(40)+j*ndf),hr(np(40)+j*ndf+1),hr(np(40)+j*ndf+2),
157       &       hr(np(43)+j*ndm)+hr(np(40)+j*ndf),    ! deformed nodal pos.
158       &       hr(np(43)+j*ndm+1)+hr(np(40)+j*ndf+1),
159       &       hr(np(43)+j*ndm+2)+hr(np(40)+j*ndf+2),
160       &       hr(np(27)+j*ndf),hr(np(27)+j*ndf+1),hr(np(27)+j*ndf+2),
161       &       hr(np(57)+numnp*1+j),                 ! principal stress 1
162       &       hr(np(57)+numnp*2+j),                 ! principal stress 2
163       &       hr(np(57)+numnp*3+j),                 ! principal stress 3
164       &       hr(np(57)+numnp*4+j),                 ! I 1
```

```
      &        hr(np(57)+numnp*5+j),                    ! J 2
      &        hr(np(57)+numnp*6+j)                     ! J 3 (v. Mises-Stress)
          end do
c         'Array 33 IX: Element Connections'
          if(nen.eq.8 .or. nen.eq.27 .or. nen.eq.20) then     !hexahedral
            do j = 0,numel-1
                write(99,2013)
      &         mr(np(33)+j*nen1),mr(np(33)+j*nen1+1),
      &         mr(np(33)+j*nen1+2),mr(np(33)+j*nen1+3),
      &         mr(np(33)+j*nen1+4),mr(np(33)+j*nen1+5),
      &         mr(np(33)+j*nen1+6),mr(np(33)+j*nen1+7)
            end do
          elseif(nen.eq.4 .or. nen.eq.10 .or. nen.eq.11) then !tetrahedral
            do j = 0,numel-1
                write(99,2014)
      &         mr(np(33)+j*nen1),mr(np(33)+j*nen1+1),
      &         mr(np(33)+j*nen1+2),mr(np(33)+j*nen1+3)
            end do
          endif !nen

        elseif((ndm.eq.3).and.(ndf.eq.6)) then !micropolar continuum elem.

          write(99,*) 'TITLE = \"FEM DATA, FEAP->TECPLOT\"'
          write(99,*) 'VARIABLES=\"X\",\"Y\",\"Z\",\"DX\",\"DY\",\"DZ\",
     &\"RX\",\"RY\",\"RZ\",
     &\"FX\",\"FY\",\"FZ\",\"MX\",\"MY\",\"MZ\",
     &\"sig11\",\"sig12\",\"sig13\",
     &\"sig21\",\"sig22\",\"sig23\",
     & \"sig31\",\"sig32\",\"sig33\",
     &\"sigc11\",\"sigc12\",\"sigc13\",
     &\"sigc21\",\"sigc22\",\"sigc23\",
     & \"sigc31\",\"sigc32\",\"sigc33\"'
          write(99,2001) numnp,numel
          if(nen.eq.8 .or. nen.eq.27 .or. nen.eq.20) then
            write(99,*) 'DATAPACKING=POINT, ZONETYPE=FEBRICK'
          elseif(nen.eq.4 .or. nen.eq.10 .or. nen.eq.11) then
            write(99,*) 'DATAPACKING=POINT, ZONETYPE=FETETRAHEDRON'
          end if !nen

          do j = 0,numnp-1
              write(99,2015)
      &       hr(np(43)+j*ndm),hr(np(43)+j*ndm+1),hr(np(43)+j*ndm+2),
      &       hr(np(40)+j*ndf),hr(np(40)+j*ndf+1),hr(np(40)+j*ndf+2),
      &       hr(np(40)+j*ndf+3),hr(np(40)+j*ndf+4),hr(np(40)+j*ndf+5),
      &       hr(np(27)+j*ndf),hr(np(27)+j*ndf+1),hr(np(27)+j*ndf+2),
      &       hr(np(27)+j*ndf+3),hr(np(27)+j*ndf+4),hr(np(27)+j*ndf+4),
      &       hr(np(58)+numnp*1+j),hr(np(58)+numnp*2+j),
      &       hr(np(58)+numnp*3+j),hr(np(58)+numnp*4+j),
      &       hr(np(58)+numnp*5+j),hr(np(58)+numnp*6+j),
      &       hr(np(58)+numnp*7+j),hr(np(58)+numnp*8+j),
      &       hr(np(58)+numnp*9+j),
      &       hr(np(58)+numnp*10+j),hr(np(58)+numnp*11+j),
      &       hr(np(58)+numnp*12+j),hr(np(58)+numnp*13+j),
      &       hr(np(58)+numnp*14+j),hr(np(58)+numnp*15+j),
      &       hr(np(58)+numnp*16+j),hr(np(58)+numnp*17+j),
      &       hr(np(58)+numnp*18+j)

          end do
```

```
c         'Array 33 IX: Element Connections'
          if(nen.eq.8 .or. nen.eq.27 .or. nen.eq.20) then     !hexahedral
            do j = 0,numel-1
                write(99,2013)
     &          mr(np(33)+j*nen1),mr(np(33)+j*nen1+1),
     &          mr(np(33)+j*nen1+2),mr(np(33)+j*nen1+3),
     &          mr(np(33)+j*nen1+4),mr(np(33)+j*nen1+5),
     &          mr(np(33)+j*nen1+6),mr(np(33)+j*nen1+7)
            end do
          elseif(nen.eq.4 .or. nen.eq.10 .or. nen.eq.11) then !tetrahedral
            do j = 0,numel-1
                write(99,2014)
     &          mr(np(33)+j*nen1),mr(np(33)+j*nen1+1),
     &          mr(np(33)+j*nen1+2),mr(np(33)+j*nen1+3)
            end do
          endif !nen

        endif !ndm.and.ndf
       endif

c      Modified filename format
1999   format('_',I4.4,'.dat')                                          *
2001   format('ZONE N=',I6,' E=',I6)
c      2D formats
2002   format(14(F16.8,4X))
2003   format(4(I6,4X))
2004   format(3(I6,4X))
c      3D formats
2012   format(18(F16.8,4X))
2013   format(8(I6,4X))
2014   format(4(I6,4X))
2015   format(33(F16.8,4x))
       end
```

Listing C.1: Modified FEAP user macro

D ParFEAP shell script

The following listing represents a shell script that has been written by the author in order to enable the execution of ParFEAP in batch mode. A summary of its features can be found in chapter 5.1. In case of an input file named *iexample*, a valid call could yield

```
./feapload.sh example 4 mg
```

which would start a parallel run on four processors using the configured options for the multigrid run. If a partitioning run should take place beforehand, this command should be altered to

```
./feapload.sh example 4 mg 1
```

regardless of which partitioning method (METIS or ParMETIS) is applied. This syntax is however only valid if used in terms of a job command file as presented in chapter 6.1.3 where the LoadLeveler launches MPI itself. In other cases, the same command has to be passed to *poe*, for example via

```
poe ./feapload.sh example 4 mg 1 -procs 4 -hostfile hostfile
```

which is particularly helpful for testing issues.

```
1  #!/bin/ksh
2  #
3  # This script should make users able to run ParFEAP in batch mode while
4  # providing some convenient features like an automatic partitioning at the same
5  # time7
6  #
7  # Usage:
8  # ./feapload feap_filename number_of_processors solver partitioning
9  # In this context, "feap_filename" refers to the definition FEAP proposes: If
10 # the standard FEAP input file is named "ifilename", then the word "filename"
11 # should be the first argument supplied to this script.
12 # The second argument represents an integer indicating the number of processors
13 # processors that should be used during parallel execution (n > 1 !).k
14 # Please make sure this number is consistent to the number of partitions
15 # specified in the FEAP input file!
16 # The third argument tells the script which solver setting presets ("sg" or
17 # "mg")to use. For the options contained in the preset, see the values of the
18 # variable "SGOPTIONS" and "MGOPTIONS"
19 # The final argument indicates whether an already partitioned geometry should be
20 # assumed (argument is 0 or not present) or a partion run should be included
21 # (otherwise)
22 #
23 # Course of actions:
24 # - If partitioning has been activated: Runs ParFEAP using the input file named
25 #   'ifilename' (partitioning); Results will be logged in 'partition.log'
26 # - Runs ParFEAP in parallel mode using the input files named
27 #   'ifilename_0001' to 'ifilename_000n' and 'solve.filename'; A log will be
28 #   stored in 'parrun.log'
29 # - create a tar-file archiving all relevant files, which are: 'ifilename',
30 #   'Ofilename', 'Lfilename', 'ifilename_0001' ... 'ifilename_000n',
31 #   'Ofilename_0001' ... 'Ofilename_000n', 'Lfilename_0001' ...
32 #   'Lfilename_000n', 'solve.filename' and the global plot files ('G...')
33 #   and _deletes_ the archived files (except of the original input file)
34 #
35 # Additional Notes:
```

```
36 # - Make sure to adjust the constants stated below according to your
37 #   requirements before using this script
38 # - This script has been tested on IBM AIX systems only!
39
40 # Constants
41 readonly MPI=/usr/bin/poe
42 readonly HOSTFILE=$HOME/bjoern_m/mpihosts
43 readonly PARFEAP=$HOME/bjoern_m/feap82/parfeap/feap
44 readonly SGOPTIONS='-ksp_type cg -pc_type jacobi -log_summary -get_total_flops
45     -options_left'
46 readonly MGOPTIONS='-ksp_type cg -pc_type prometheus -pc_mg_type multiplicative
47     -aggmg_smooths 1 -prometheus_mis_levels 2
48     -prometheus_random_mis -log_summary -get_total_flops -options_left'
49
50 # Assembles the suffix of a parallel file (e.g. "0001" for the first file or
51 # "0016" for the 16th file)
52 getSuffix ()
53 {
54   id=$1
55
56   infix=''
57   case $id in
58     [1-9])
59       infix='000'
60       ;;
61     [1-9][0-9])
62       infix='00'
63       ;;
64     [1-9][0-9][0-9])
65       infix='0'
66       ;;
67     [1-9][0-9][0-9][0-9])
68       ;;
69     *)
70       echo 'Invalid file id'
71       exit 5
72       ;;
73   esac
74
75   echo "${infix}${id}"
76 }
77
78 # Check appropriate number of arguments
79 if ([ $# -lt 3 ] || [ $# -gt 4 ])
80 then
81   echo "Usage: $0 feap_filename number_of_processors sg/mg [do_partitioning]"
82   exit 1
83 fi
84
85 # The first argument is the filename. Existance will be checked later
86 filename="$1"
87
88 # Second parameter has to be an integer
89 # Would have used 'declare -i nproc="$2"' but doesn't work on AIX!
90 nproc="$2"
91 if [ $nproc -le 1 ]
92 then
93   echo 'Number of processors has be greater than 1'
```

```
94    exit 2
95  fi
96
97  # Third parameter has to be either 'sg' (single grid) or 'mg' (multigrid)
98  options=' '
99  if [ $3 == 'mg' ]
100 then
101   options=$MGOPTIONS
102 elif [ $3 == 'sg' ]
103 then
104   options=$SGOPTIONS
105 else
106   echo "Third parameter has to be either 'mg' or 'sg'"
107   exit 3
108 fi
109
110 # Fourth parameters tells us whether to start a serial run (partitioning run)
111 doPartitioning=0
112 if ([ $4 ] && [ ! $4 == 0 ])
113 then
114   doPartitioning=1
115 fi
116
117 serial_ifile="i${filename}"
118 parallel_ifile="i${filename}_0001"
119
120 if [ $doPartitioning == 0 ]
121 then
122   echo 'No partitioning requested'
123 else
124   # Check if $serial_ifile exists
125   if [ ! -r $serial_ifile ]
126   then
127     echo "Invalid filename '${serial_ifile}'. File does could not be read"
128     exit 4
129   fi
130
131   echo 'Starting partitioning'
132   `rm -f feapname`
133   $MPI $PARFEAP -procs 1 -hostfile $HOSTFILE > partition.log <<EOT
134 $serial_ifile
135
136
137
138
139 y
140 EOT
141 fi
142
143 # Check if all parallel input files exist
144 i=1
145 while [ $i -le $nproc ]
146 do
147   file="i${filename}_`getSuffix ${i}`"
148   if [ ! -e $file ]
149   then
150     echo 'Partitioning failed'
151     exit 4
```

```
  fi
  (( i+=1 ))
done

# Execute ParFEAP while submitting some commands to stdin
echo 'Starting parallel run'
`rm -f feapname`
$PARFEAP $options > parrun.log <<EOT
$parallel_ifile




y
EOT

# Save results in archive file (the suffix is UNIX timestamp) and delete
# archived files
now=`date +%s`
tarfilename="${3}run${now}.tar"
echo "Starting backup to ${tarfilename}"
`tar -cf $tarfilename "solve.${filename}" partition.log parrun.log`
`rm partition.log parrun.log`
for filetype in i L O
do
  serialfile="${filetype}${filename}"
  `tar -uf $tarfilename $serialfile`

  # Do not delete the original input file ('ifilename') but all the others
  if [ $filetype != i ]
  then
    `rm $serialfile`
  fi

  # Take care of partitioned input/output/log files
  i=1
  while [ $i -le $nproc ]
  do
    parfile="${serialfile}_`getSuffix ${i}`"
    `tar -uf $tarfilename $parfile`
    `rm $parfile`
    (( i += 1 ))
  done
done

# Archive global plot files (files for stresses (s), principal stresses (p) and
# dispalacments (d)) too.
for plottype in s p d
do
  # Each job creates it's own plot files; get them all
  i=1
  while [ $i -le $nproc ]
  do
    # There can be multiple plots of the the same type (e.g. displacements in
    # three directions) which are labeled in order of the execution of the
    # solution command
    n=1
    plotfile="G${filename}_`getSuffix ${i}`.${plottype}0001"
```

```
210     while [ -e $plotfile ]
211     do
212       `tar -uf $tarfilename $plotfile`
213       `rm $plotfile`
214       (( n += 1 ))
215       plotfile="G${filename}_`getSuffix ${i}`.${plottype}`getSuffix ${n}`"
216     done
217     (( i += 1 ))
218   done
219 done
```

Listing D.1: ParFEAP shell script